A Descriptive Language for Information Object Reuse through Virtual Documents

Anne-Marie Vercoustre[1] and François Paradis
CSIRO Mathematical and Information Sciences
723 Swanston Street, Carlton,
3053 Victoria, Australia
{Anne-Marie.Vercoustre, Francois.Paradis}@cmis.csiro.au

Abstract

The importance of reuse is well recognised for electronic document writing. However, it is rarely achieved satisfactorily because of the complexity of the task: integrating different formats, handling updates of information, addressing document author's need for intuitiveness and simplicity, etc. In this paper, we present a language for information reuse that allows users to write *virtual documents*, where *dynamic* information objects can be retrieved from various sources, transformed, and included along with *static* information in SGML documents. The language uses a tree-like structure for the representation of information objects, and allows querying without a complete knowledge of the structure or the types of information. The data structures and the syntax of the language are presented through an example application. A major strength of our approach is to treat the document as a non-monolithic set of reusable information objects.

1 Introduction

Information reuse has a long established intellectual heritage encompassing relational databases, text-mark-up, record management systems and object-oriented thinking. To a great extend, the work of producing new documents involves reusing pieces of previously existing documents [18], modifying them and assembling them in a new way, according to the purpose of the new document.

The widespread adoption of electronic documents should allow for more information reuse. However it is often hard to put in practice, because the documents are not defined at the level of granularity and modularity needed for information reuse. Even with standards such as SGML [10], the inclusion of elements from one document to another can be very difficult, because the elements are strongly typed and there could be incompatibilities between the DTDs (document grammars) [5].

As the Web becomes a major source of information, a lot of interest has arisen, not only for searching for information, but for reusing this information in new pages, or directly from applications. Unfortunately HTML tags do not provide a significant level of structure for identifying and extracting information, since they are mostly used for presentation issues. A solution is to build a model for a set of homogeneous pages using a structuring schema [3,6].

Updating information within documents also requires a lot of effort: it is very costly to create a new version of the full document each time the information has changed or new information has been added. The difficulty is to make these changes consistently throughout an entire documentation. A solution to this problem is to use

[1] This author's permanent position is INRIA-Rocquencourt, France.

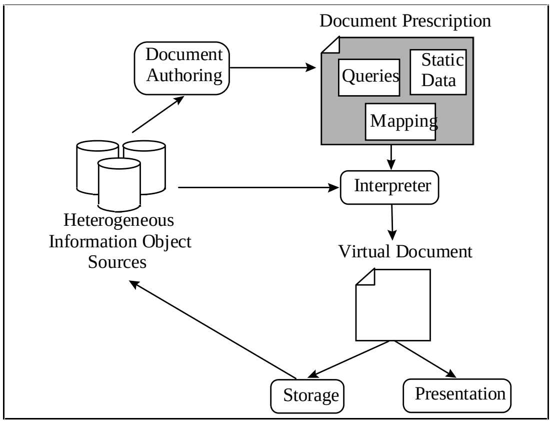

Figure 1: RIO Architecture

virtual documents: that is, documents that are dynamically generated on demand and therefore always reflect the new states of the world. Unfortunately, writing the "instructions" to generate a virtual document is a tedious process that often requires more programming skills than document writing.

This paper introduces a new language for writing virtual documents, which allows the creation of new documents by extracting the relevant information from existing documents, and assembling it into a new document. Three important features of the language are the following:

- *descriptive syntax*. The focus in our language is on the information objects rather than on the procedures as in script or command-based languages.
- *heterogenous information*. The information that can be inserted in a virtual document can come from various sources and in different formats: tables from SQL, objects from OQL, semi-structured data from HTML pages, etc. Our language proposes a tree-like representation of all information objects.
- *mapping*. We propose mapping rules in order to include the "dynamic" information of the document in the appropriate DTD.

We first discuss the context of our work and present an example of its application. We then give a brief overview of the tree-like structures used by the language, before presenting the language itself. We conclude with a comparison with other approaches and further work.

2 The RIO project

The RIO (Reuse of Information Object) project [23] aims to develop techniques which can support information reuse in various contexts. The focus of the project is currently on the reuse of structured information from heterogeneous sources, including OO or relational databases, SGML and HTML documents, and possibly any kind of *semi-structured* data [1].

Our approach is to introduce a *document prescription* of the virtual document to be generated. A document prescription consists of:

- Static data, that is the structure and the text that doesn't change in the document.
- Queries, or the commands needed to generate the dynamic part of the document.

```
<!DOCTYPE  project [
<!ELEMENT project  --  (summary, attractiveness, feasibility, priorities, customers,
scope)>
<!ATTLIST  project    gate NUMBER  "0"
                       idID   #IMPLIED
                       class   (internal, CRC, industrial)                >
<!ELEMENT summary  --  (header, description+, outcome+, actions*)  >
<!ELEMENT header oo (title, author, date, portfolio, business)            >
<!ELEMENT description -o  (#PCDATA)                                       >
<!ELEMENT outcome -o  (#PCDATA)                                           >
<!ELEMENT action -o  (#PCDATA)                                            >
…
]>
```

Figure 2: DTD for a project document

- Transformation instructions, to convert the reused information objects into new document objects.

The overall architecture of the system is described in Figure 1. In order to produce a virtual document, the interpreter of the document prescription executes the following tasks: send the "native" queries to the database server and get the results back; assemble the results into a generic tree representation; select, join and transform objects in the tree model; and finally, map the selected objects into the document structures of the target virtual document.

In the following section we give an example of a virtual document that we will use along the paper to present our generic data model as well as the prescription language.

3 An example

Our example is a yearly activity report for our EDC (Electronic Document and Commerce) group. The report contains the list of people who belong to the group, a description of the various projects they have been involved in the current year, as well as their collective list of publications. The existing sources of information are:

- an SQL database of persons, with names, title, location, portfolio; this database is primarily used for generating our internal and external Staff Web pages.
- An OO database of SGML documents that contains a short description (2 to 6 pages) of the current projects (research or industrial). These documents conform to the (incomplete) DTD shown in Figure 2. Let's call them *project* documents. The project documents are stored in the OO database using the same kind of mapping between the DTD and the database classes as described in [8].
- A database of bibliographic references to our external publications. This database can only be accessed through the Web, which means that the available source of information is an HTML page of bibliographic references.

The activity report is generated by combining the following information: the EDC employees from the SQL database, project descriptions from excerpts of the project documents, and publications from the HTML pages, using a join with the EDC employees.

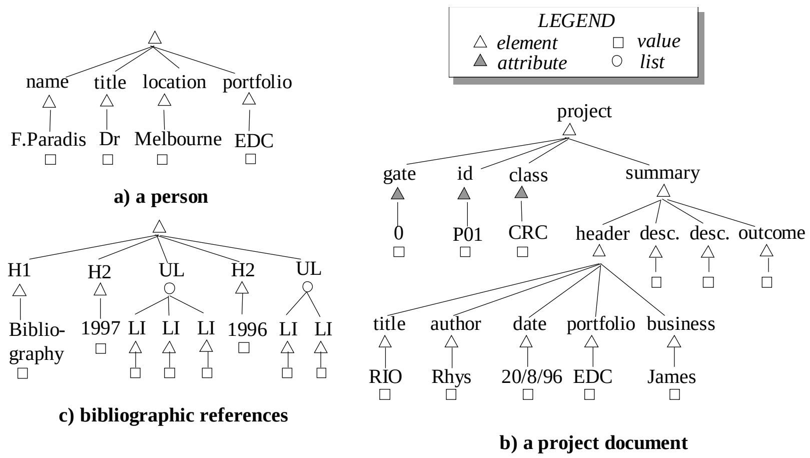

Figure 3: Tree representation examples

Section 4 will present our language with detailed examples for building the parts of the activity report. But before introducing our language, we need to present the tree data model used for integrating the answers from the various databases or other information sources.

4 The tree model

We introduce our integration model which is very similar to those presented in [1,3]. As in these approach we are not using a model that encompasses all the source models. Instead we use a generic and minimalist approach. The data structure we use consists of ordered labeled trees. Each tree node can have one of the following type:

- a node of type "value" has no children. Its label is the appropriate atomic value.
- a node of type "element" has one child (of any of the four types) for each of its component. Its label is the name of the element.
- a node of type "attribute" has one child of type "value" that contains the value of the attribute. Its label is the name of the attribute.
- a node of type "list" has as many children as elements in the list.

Figure 3 shows examples of tree representations for the objects of our example application.

SQL tables are represented with element nodes, with an unlabeled node for each row, which has the columns as children. Of course this mapping is not unique; for an alternative see [3,p.2]. Figure 3a gives an example of a person from our STAFF database. The full table would be represented by a list of persons.

SGML elements are represented by element nodes with a label corresponding to the tag name. These nodes have one child for each element content. SGML attributes are represented by an attribute node under the element node. As lists play a very strong role in documents (in contrast with standard databases) we also retain the information that an element corresponds to a list. Figure 3b gives an example of a project document from our Project database.

HTML elements are represented by a simplified version of their parsing tree, in the same manner than for SGML elements. However a typical HTML tree will be more "flat" than an SGML one, as it lacks the logical structure of the document. Figure 3c gives an example of a our publications database, which is stored as one HTML file, containing several lists of publications organised by year.

Unlike conventional object-oriented techniques, the elements in our trees are untyped. This doesn't have an appreciable impact on global efficiency as most of the retrieval is performed at a lower level by the native queries. Our structures can be considered like parse trees, where several nodes can have the same label.

It must be noted here that objects can form cyclic graphs, so we might need to represent links between nodes using an object identifier. However since our current focus is on SGML documents rather than general object-oriented databases, we only need to consider links between documents or between parts of document. In SGML these links are purely syntactic, so we can keep them in the tree as they are, ie. by their SGML tags.

5 The prescription language

In order to write the document prescriptions and to query the tree structures described above, a *prescription language* was designed. In this section we describe in details the two main components of the language: query and transformation instructions. We then conclude with an example that shows how these are included in a document along with static information.

5.1 Queries

A key issue for the query component of the prescription language is to maintain the diversity and flexibility of the underlying systems, while dealing with information that is quite different in nature. Our approach is to allow queries to be expressed in the native database language, but to supplement them with our own search mechanism to select and combine the results. A typical statement, to retrieve employees from the STAFF SQL database who have authored a project document in the PROJ OQL database, would be as follows:

```
pick $e
from sql(select * from STAFF) as $e,
     oql(select p.summary.header from Proj p) as $h
where $e.name = $h.author
```

The language can be roughly divided into four parts: the selection of information, the construction of new information, the querying commands, and the assignments. We now present those four parts with examples.

5.1.1 Information selection

An important feature of the language is the ability to select any part of a query result. Given our uniform tree-representation of all information, this sums up to selection of nodes in a tree. Our approach is based on the idea of *generalised path expressions* [8], which permit browsing a tree without a complete knowledge of its structure.

The *dot* operator (".") is defined for accessing the direct children of a node using their label, while the *dot-dot* operator ("..") is used to recursively access the direct and indirect children of a node. Assuming for example that $d denotes a project document[2], then the following are valid selection expressions:

1.1)	$d.summary	*selects the summary*
1.2)	$d.summary.description	*selects all summary descriptions*

[2] See section 5.1.4 below for making assignments to identifiers.

1.3)	$d..description	*selects all descriptions*

These expressions actually return *all* the matching subtrees - in a list if there is more than one element - unlike other OO approaches where there can only be one possible match (as is the case with OQL for example). As descriptions can only occur directly under summaries, query 1.3 is equivalent to 1.2. Supposing that descriptions could be nested, then query 1.3 would return all descriptions whatever their level. If this is not desirable, a more strict selection can be achieved with the dot operator.

One can also use the node types for selection. For example:

1.4)	$d.#ATTR	*all attributes (gate, id, class)*
1.5)	$d.id#ATTR.#VALUE or $d.id.#VALUE	*id value*

Here #ATTR and #VALUE refer to the attribute and value node types, respectively.

Children can be selected according to their rank using the "[]" operator in OQL fashion. The children are then considered as a list starting at position 0. The #FIRST and #LAST keywords can be used to refer to the first and last elements, respectively. For example:

1.6)	$d.summary.[#FIRST]	*selects the summary header*
1.7)	$d.#ATTR[0]	*get the first attribute*
1.8)	$d..description[#FIRST]	*selects the first description*

The range (":") and union ("&") operators are defined for multiple selections. For example:

1.9)	$d.#ATTR[#FIRST:#LAST]	*all attributes; equivalent to 1.4*
1.10)	$d.summary&feasability	*the summary and the feasability*
1.11)	$d.summary:feasability	*the summary, attractiveness and feasability*
1.12)	$d..header&description[0]	*the header and first description*

If no element is found, the selection will return an empty list.

5.1.2 Information construction

It is possible to construct nodes of a given type, using the list, elem and attr functions. For example:

2.1)	list($d..title,$d..author) or ($d..title,$d..author)	*a list node with title and author*
2.2)	elem(paragraph, $d..description)	*element node with label "paragraph" and with descriptions as children*
2.3)	attr(version,$d.id.#VALUE)	*attribute node with label "version", and document id as child (value)*

In our language, strings are considered as value nodes with the actual characters stored in the label. The str function can be used to construct a string from a tree, concatenating all "content" nodes (ie. value nodes that are not the child of an attribute). When passed several arguments, it also concatenates them. For example:

2.4)	str($d) or str($d..#ELEM.#VALUE)	*content of the whole document*
2.5)	str($d.summary,$d.scope)	*summary and the scope*

The type and label of a node can be accessed with the corresponding functions.

2.6)	type($d.id)	*the string "#ATTR"*
2.7)	label($d.id)	*the string "id"*

Finally, information can be added or removed from a tree with adopt and excluding, respectively. For example:

2.8)	$d **excluding** priorities	*the document without priorities*
2.9)	$d **adopt** attr(version,1.0)	*the document with version*

5.1.3 Querying

Native queries can be expressed with the corresponding function. For example:

3.1)	`sql(select * from STAFF)`	*select all records from STAFF SQL database*

Queries on our tree representation follow the syntax "pick…from…where" (the from and where clauses being optional), which is similar in style and semantics to SQL "select…from…where". For example:

3.2)	**pick** $d..description or **pick** description **from** $d	*select all descriptions from $d* *(equivalent to 1.3)*
3.3)	**pick** $p **from** oql(select project from Proj p) **as** $p **where** $p..portfolio = "EDC"	*select project documents that* *belong to the "EDC" portfolio"* *from the Proj OQL database*

Note that selection and construction operations are also interpreted as valid queries; so all examples of the previous sections are also queries.

5.1.4 Assignments

Query results can be saved using an *assignment* expression. An *identifer* is used to refer to them. For example:

4.1)	**define** $e **as** sql(select * from STAFF); $e.name	*defines $e as all employees* *get all employees names*

5.2 Mapping

The purpose of the mapping instructions is to convert elements from the generic tree representing the answers to queries, into elements of the virtual document. In particular, the mapping instructions must permit these various SGML constructions: PCDATA, attributes, aggregates, recursive lists.[3] In addition, often-used constructions, such as tables (a common example of a recursive list), must be easy to express.

Our approach is to keep all the SGML-dependent syntax in the *static* part of the document prescription, and to define the mapping instructions as part of SGML elements in the document. The mapping instructions are written as SGML processing,

[3] Choice (ie. alternatives) are not to be considered since we are dealing with documents, not with DTDs.

and always refer implicitly to the tag that includes them (which we call the *including tag*). For example:

```
<title><?processing instruction></title>
```

We now see in details the mapping operations through the various SGML structures that can be built.

5.2.1 Text

Query statements are mapped into text. The processing instruction:

```
<?$q>
```

is interpreted as text($q), which is a function similar to str, except that it inserts blanks between elements. Multiple queries are treated as lists. The interpretation of:

```
<?$q1; …; $qn>
```

is given by text($q1,…,$qn).

5.2.2 SGML trees

In many cases, we are not only interested in the textual content of the query results, but also in the actual structure of the tree. The sgml function can be used for generating an SGML fragment of document that maps exactly the tree structure. The straight mapping of a tree into an SGML document involves a pre-order tree traversal, using the element labels as the tags and their value as the content.

The sgml function can be very useful to simplify the mapping language by first building the appropriate tree using the construction language, eventually adding, removing or modifying the elements. It also makes it easy to include part of existing document into a document with the same DTD (eg. HTML/HTML). More complex library functions could be implemented, possibly in DSSSL, for other standard transformations (eg. Article to HTML).

5.2.3 Lists

The map instruction is used to create a sequence of SGML elements from the results of a query. An expression like <?map $var in $list><tag>…</tag>, is replaced by the interpreter by as many tag elements as there are elements in $list.[4] Within the tag element, $var can be used to refer to the current element in the list. For example:

```
<?map $name in list("François","John")><LI>my name is <?$name></LI>
```

produces:

```
<LI>my name is François</LI>
<LI>my name is John</LI>
```

Another form of the map allows an *index variable*, to refer to the current element number. For instance to add a counter in the previous list, one could do:

```
<?map $name[$i] in list("François","John")><LI>(<?$i>) My name is <?
$name></LI>
```

5.2.4 Tables

Complex structures like tables can be built using map instructions. For example, given the following query, which retrieves the personnel involved in the "EDC" portfolio:

```
<?define $staff as (pick * from
    sql(select * from STAFF where portfolio="EDC")>
```

[4] The syntax <?map $var in $list><?begin>…<?end> is also provided to repeat elements that are not properly enclosed within tags.

then the following prescription would output a table where the rows are those employees, and the columns their name and title.

```
<TABLE>
    <TR><TD>name</TD><TD>title</TD></TR>
    <?map $s in $staff><TR>
            <TD><?$s.name></TD> <TD><?$s.title></TD></TR>
</TABLE>
```

The above example can be rewritten using nested maps as follows:

```
<TABLE>
    <TR><?map $field in unique
$staff.name&title><TD><?label($field)></TD></TR>
    <?map $s in $staff><TR>
            <?map $ss in unique $s.name&title><TD><?$ss></TD></TR>
</TABLE>
```

This sort of construct can be particularly useful when the structure of the records is not known. Removing ".name&title" in the example above would produce an employee table with all their fields.

5.2.5 Attributes

The attributes instruction is used to add dynamic attributes to tags. It takes as an argument either an attribute node or a list of attribute nodes. For example, the following:

```
<A><?attributes attr(href,"#sect1")>section 1</A>
```

produces <A href=#sect1>section 1</A>. When attribute values can be taken directly from trees, the syntax is still more immediate. Suppose for example that $s refers to some HTML element with attribute name equals to sect1. Then the following:

```
<A><?attributes $s.name>…</A>
```

would produce <A name=sect1>…</A>.

5.2.6 Links

A typical example of the use of internal links is the creation of a table of content at the beginning of an HTML page, with links to further sections. It can be implemented in three steps: first the retrieval and construction of the sections, second the construction of the table of contents, and third the inclusion of the sections in the document. Assuming $proj refers to a list of projects documents, as defined in section 3, then the sections can be constructed as follows:

```
<? define $sections as
    (pick $s adopt attr(name,idname(sect))
     from $proj.summary )>
```

As the sections are retrieved, they are added (with the adopt operation) a name attribute that will be used for referencing them. The function idname is used for generating label names; it returns a new identifier with its string parameter each time it is called (here: sect1, sect2, …).

The table of contents consists of a list of pointers to the sections: it can be built by mapping every section title into a <LI> element, and adding an href attribute that points to the corresponding section.

```
<UL>
    <?map $s in $sections><LI>
       <A><?attributes attr(href, str("#", $s.name.#VALUE))>
       <?$s.header.title ></A></LI>
</UL>
```

```
<HTML>
<?define $staff as sql(select * from STAFF where portfolio="EDC")>
<?define $proj as oql(select p from Proj p where p.portfolio="EDC")>
<?define $pub as url(http://www.mel.dit.csiro.au/pubs/pubs.html)>
<H1>Activity report for 1997</H1>
<H2>Participants</H2>
<UL><LI><?map $staff.name></LI></UL>
<H2>Projects</H2>
<UL><?map $p in $proj><LI>
      <H3><?$p.summary.header.title></H3>
      <P><?$p.summary excluding header></P></LI>
</UL>
<H2>Publications</H2>
<P><?pick $onepub from $pub.H2 as $header[$i],
                    $pub.[$i+1].LI as $ onepub,
                    $ staff.name as $name
    where $header contains "1997", $ onepub contains $name>
</P>
</HTML>
```

Figure 4: The yearly activity report prescription

The str function concatenates the strings like the text function, but without adding extraneous spaces.

Finally the sections can be included in th e document with the following:

```
<?map $s in $sections><A><attributes $ s.name>
    <h2><?$s.header.title></h2>
    <P><?$s.description></P></A>
```

Here each section consists of an anchor element (<A>), with a header (<H2>) and description paragraph (<P>).

External links can be built the same way using the tags provided by the DTD.

5.3 A complete example

Figure 4 shows the prescription for the activity report of the EDC portfolio for year 1997, in HTML format. The document consists of some initialisation statements, which retrieve information from the various databases, followed by three sections: the participants, the projects, and the publications. The first two sections are taken directly from the sql and oql queries: the participants are the list of EDC staff members, and the projects is the list of items containing the projects title and their summary. The publications section is more complicated to construct. We first find the 1997 publications in the “flat” HTML structure, and then only keep those containing a participant name.

The pick statement in the publications section requires a bit more explanation. The expression “$pub.H2 as $header[$i]” matches $header with an H2 element (which is further restricted in the where clause to contain “1997”), and $i with its rank. The expression “$pub.[$i+1]” matches the element immediately following $header, which happens to be a list where every item is a publication ($onepub). Finally, only publications containing a member of the staff are kept.

6 Related work

Other works have looked at querying structured or semi-structured information, either from a database point of view, as with OQL [21], or from a document processing point of view, as with HyQ [13]. We have already mentioned that our syntax is quite similar to OQL. This language, however, is inadequate for “loosely” structured queries. In order to issue query 1.3 in OQL, one would have to define a method to recursively access the `description` fields of the children nodes. HyQ does not provide any primitive for constructing information, furthermore, it tends to be complex and not very intuitive. Consider for example the formulation of query 1.3 in HyQ:

```
<HyQ qdomain=$d>
Select(DOMTREE Eq(Proploc(CAND GI) "description"))
</HyQ>
```

Our selection of objects is similar to the generalised path expressions found in POQL [2]; however, the interpretation is simplified since our tree structures are untyped rather than being constrained by a database schema. This need to query semi-structured data without strict typing conventions has been recognised in Lorel [4], which use coercion between types to address the problem. An alternative to path expressions is tree patterns, as in SgmlQL [16]. SgmlQL actually goes beyond the querying, with primitives to construct new trees from existing ones, however, as any of these languages, it does not address the problem of conversion to a different DTD.

DTD conversion has been addressed already by a number of approaches: either by general-purpose languages such as DSSSL[14], or within SGML editors for supporting cut-and-paste between incompatible elements [5]. Other languages for restructuring information, like Lorel, focus on the integration of heterogeneous data structures or translation from one world to another, including from/to a document format. These languages, as well as DSSSL, could be used for implementing library *coercion* functions to map complex or frequently used structures, but are beyond the reach of document writers because of their complexity,.

Various languages for querying the Web in a database style have been developed [6,15,19]. They allow searching for specific elements in pages, as well following links, but they do not support querying the full SGML structure.

Languages for building documents from existing sources of information have been less explored.

SGML offers a static way of assembling entities from external files. All the assembled parts must conform to the DTD of the embedding document and there is no language for dynamic transformations. [22] enlightened the need for assembling dynamically generated parts of documents, and suggested using the *process instruction* mechanism. Our language precisely defines a language for building the virtual components, to be embedded in process instructions, then providing an SGML-compliant solution.

For reusing Web pages, Araneous [6] offers a limited model for creating new Web pages after querying existing ones. The constructed pages can only include text, lists and links to other pages, and not the full-range of HTML constructs; from the definition of new pages, a new set of *static* HTML pages are created.

Various ways of including pieces of information within a static HTML page exist, but HTML itself does not offer much more than <IMG> tags, and a server side *include*. Advanced servers, like PHP [17] or w3-msql [11] offer interface for accessing SQL databases from HTML. This works by embedding SQL queries and control instructions in HTML comments that are pre-processed on the server. Their script-based approach have the disadvantage of mixing the commands with the HTML syntax, which makes it difficult to update them or to include them in other documents. They do not allow for merging results from various databases.

MacWeb [20] synthesises documents by dynamically combining fragments of information that are organised in an object-oriented knowledge base. It uses a script language for describing the structure and the semantic content of the virtual document, which makes it difficult to author the document specification.

7 Conclusion

Reusing information contained in documents is becoming a major issue, whether it is proprietary information or information available from the Internet. In this paper we have presented a language for reusing information objects from heterogeneous sources. Our approach is to use a *middleware* format to integrate the results of queries to the various sources and to map its into a new (virtual) document.

The virtual document prescription is document-centred, which allows for authoring virtual document in a more descriptive ways than using a program or a scripting language. The prescription can also be viewed as a configuration specification for the virtual documents (a "makefile" for document), that can be stored as a document.

The language is designed primarily for reusing information objects from databases and structured documents. It works also with HTML documents, but the HTML tags may not be the right granularity for information objects. If XML [7] is adopted, it will provide the explicit structuring schema for Web pages for extracting objects.

Further extensions to the language under considerations include:

- adding parameters to the language. A virtual document prescription would then act as a generic template for a variety of virtual documents.
- partial instantiation of the virtual document prescription for document edition, or for efficiency.
- control instructions: The virtual document construction may depend upon the actual results of queries (eg. no answer or to many answers).

We are also considering extending an SGML editor to support the creation of virtual documents.

References

[1] S. Abiteboul, "Querying semi-structured data", ICDT (invited talk), 1997.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G.Moerkotte, and J. Simeon, "Querying documents in object databases", *Journal of Digital Libraries*, 1997. To appear. http://www-db.standford.edu.

[3] S. Abiteboul, S. Cluet, and T. Milo. "Correspondence and translation for heterogeeous data", in *Proc. ICDT 97.*

[4] S. Abiteboul, D. Quass, J. McHugh, J.Widom, and J. Wiener. "The Lorel query language for semi tructured data", *Journal of Digital Libraries*, 1996. Ftp://db.standford.edu/pub/papers/lorel96.ps

[5] Akpotsui, V. Quint, C. Roisin, "Type Modeling for Document Transformation in Structured Editor", *Mathematical and Computer Modelling*, 1994.

[6] P.Atzeni, G. Mecca, P. Merialdo, and E. Tabet, *Structure in the Web*, Technical Report 19-96, Dipartimente di Informatica e Automazione, Universita' di Roma, 1996.

[7] T. Bray, and C.M. Sperberg-McQueen, *Extensible Markup Language (XML*), W3C Working Draft Nov 96, available from http://www.w3.org/pub/WWW/TR/WD-xml-961114.html

[8] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, “From Structured Documents to Novel Query Facilities”, in *Proceedings of the ACM SIGMOD Conference on Management of Data*, Minneapolis, Minnesota, pp313-324, 1994.

[9] Steven J. DeRose and David G. Durand, *Making Hypermedia Work: A User’s Guide to HyTime*, Kluwer Academic Publishers, 1994.

[10] C.F. Golfarb, *The SGML Handbook*, Clarendon Press, Oxford, 1990.

[11] Hughes Technology, *W3-mSQL & Lite Library*, http://Hughes.com.au/library/lite/

[12] ISO 8879. *Information Processing-Text and office systems-Standad Generalized Markup Language* (SGML), 1986.

[13] ISO 10774. *Information Technolgy- Hypermedia/Time-based Structuring language* (HyTime), 1992.

[14] ISO 10179. *Document Style and specification Language* (DSSSL), 1996.

[15] D. Konopnicki and O.Shmueli, “W3QS: A query System for the World Wide Web”, in *VLDB*, 1995.

[16] J.Le Maitre, E. Murisasco, M. Rolbert, “SgmlQL, un langage d'interrogation de documents structurés”, *Proceedings of BDA'95*, Nancy August 95 (*in French*). See also *SgmlQL reference manual* (*in English*) at: http://www.lpl.univ-aix.fr/projects/multext/MtSgmlQL /MQL2.html

[17] R. Lerdorf, *PHP/FI Documentation*, http://www.vex.net/php/doc.phtml

[18] D.M. Levy, “Document reuse and Documents systems”, *Electronic Publishing*, Vol.6(4) pages 339-348, December 1993.

[19] A. Mendelzohn, G. A. Milhaila, and T. Milo, “Querying the World Wide Web”, *PDIS*, 1996, available at ftp: ftp.math.tau.ac.il/pub/milo/ websql.ps

[20] J. Nanard, M. Nanard, “Using Types to incorporate Knowledge in Hypertext”, *Proceedings of the 3rd ACM Conference On Hypertexts*, ACM Press, San Antonio (Texas), pp. 329-344, December 1991.

[21] ODMG. “Object Query Language”, draft version of chapter 7 to appear in *The Object Database Standard: ODMG V2.0*, 1997.

[22] D. Skar, “Gratuating from File-based to Info-based Document Construction”, *SGML Asia Pacific Conference*, Sydney, September 1996.

[23] A.M. Vercoustre, J. Dell’Oro, and B. Hills, “Reuse of Information through virtual documents”, *Australian Document Computing Symposium, Melbourne*, April 97.