A Language for Publishing Virtual Documents on the Web

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

Abstract

The Web is creating exciting new possibilities for direct and instantaneous publishing of information. However, the apparent ease with which one can publish documents on the Web hides more complex issues such updating and maintaining Web pages. We believe one of the crucial requirements to document delivery is the ability to extract and reuse information from other documents or sources. In this paper we present a descriptive language that allows users to write *virtual documents*, where *dynamic* information can be retrieved from various sources, transformed, and included along with *static* information in HTML documents. The language uses a tree-like structure for the representation of information, and defines a database-like query language for extracting and combining information without a complete knowledge of the structure or the types of information. The data structures and the syntax of the language are presented along with examples.

1 Introduction

In recent years the Web has grown from a small, academic base, to a large network covering almost every business sector and leisure activity. In parallel to this increase of volume, the following needs have emerged for publishing information on the Web:

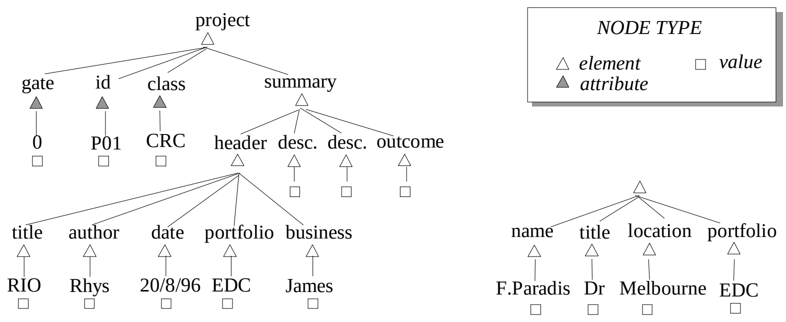
- *publishing non-HTML information*, such as from relational and object-oriented databases, word processors, etc. HTML can be - and is currently - used for purposes other than publishing, but it will not replace other formats for data processing.
- *including dynamic information*.[1] It can be very costly to update or create a new version of the full document each time the information has changed or new information has been added.
- *extracting fragments of information*. The work of producing new documents largely involves reusing pieces of previously existing documents [Lev93], modifying them and assembling them in a new way, according to the purpose of the new document. Unfortunately many formats - such as HTML - do not provide a significant level of structure for identifying and extracting fragments of information.
- *integrating heterogeneous information*. Documents can integrate information from various sources and in different formats. Rather than a simple inclusion, information will sometimes combine in complex manners to form new entities.

These needs can be addressed by *virtual documents*, ie. documents that are dynamically generated on demand and therefore always reflect the new states of the world. Virtual documents have traditionally been implemented on the Web using a programming approach, with CGI scripts or advanced server languages such as PHP [Ler] or w3-msql [Hug]. Besides being tedious and non-intuitive for the document writer, this approach makes information management more difficult, as it hides document links and dependencies behind programming code.

We propose a descriptive language for writing virtual documents, which allows the extraction and combination of information from various data sources using an SQL-like language, and can modify and integrate this information with HTML tags. This language is part of RIO (Reuse of Information Objects): a project which promotes the idea of reusing of information in the document delivery process (see [VDH97, VP97] for a description of RIO).

We first describe the data model used to integrate the various data sources (section 2), give a brief overview of the language (section 3) and a complete example of a document prescription (section 4).

[†] This author's permanent position is INRIA-Rocquencourt, France.
[1] The term "dynamic information" is also used in the literature for *interactive* documents (eg. dynamic HTML). We restrict its usage here to characterise information that can change over time.



a) a project document (SGML)

b) a person (relational database)

Figure 1: Tree representation examples

We conclude with a comparison with other approaches (section 5) and a general discussion of the context of this work (section 0).

2 The Tree Model

Our language represents heterogeneous information from data sources such as relational or object-oriented databases, SGML or SGML derivative[2] documents, etc., in a unified data model. Our model is very similar to [ACM97], in that we use a generic and minimalist approach instead of trying to encompass all the source models.

Our data model consists of trees, a structure we think is particularly appropriate in a document-oriented approach such as ours. Trees are made of nodes. Nodes can have a *label* and *children*; their semantics depend on the node type. There are four node types:

- *ELEM*. An element node is a labelled, composite entity, whose children represent structural components. For example, for an SGML element, the label is the tag name, and the children the elements contained in it.
- *VALUE*. A value node is some text (stored in the node's label). It cannot have children. In SGML, this corresponds to PCDATA.
- *ATTR*. An attribute node gives an attribute for an ELEM node. Its label is the name of the attribute, its child (children) give the value(s) of the attribute (a VALUE node). This corresponds to the element attributes in SGML[3].
- LIST. A list node is an unlabelled, composite entity, whose children represent the list elements.

Figure 1 gives two examples of representation using our data model. An example of an SGML document is shown in a). The document is a *project* consisting of a *summary* section, which is made up of subsections. The project document has three attributes: its *gate* (an indication of the current status of the project), *id* and *class*.

Shown in b) is an excerpt of the result of an SQL query over a database of persons. Viewed as a table, the elements *name, title, location* and *portfolio* would correspond to the columns, and the value nodes ("F.Paradis", "Dr", "Melbourne", "EDC") would make up one row. In general, SQL queries return multiple answers (rows), which are represented in our data model as a list, where each child correspond to an answer.

3 The Prescription Language

In our approach, virtual documents consist of *static* data, ie. information that does not change, and *dynamic* data, ie. information extracted (and possibly transformed) from external data sources. The document containing the instructions to generate the virtual document is called the *document*

---

[2] Such as HTML or XML.

[3] The storage is quite different however: the only thing that distinguishes an element attribute from the "compositional" children in our structures is the node type.

prescription. Static data is expressed in the document prescription with the usual HTML tags. Dynamic data is expressed using the *prescription language* within an SGML processing instruction. For example, the following excerpt from a document prescription contains static data (“”, “”) and a prescription instruction (“url(...)”):

```
<strong><?url(“http://www.csiro.au”)></strong>
```

In this section we give a brief overview of the *prescription language*. We first describe in 3.1 how to generate dynamic information by querying data sources, extracting and modifying results, and creating new structures. We then show in 3.1.5 how to integrate dynamic information into the virtual document. For a more detailed description of the language and complete examples of its application, see [VP97, Par97].

3.1 Generating dynamic information

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 would be as follows:

```
pick $e
from query(“jdbc:msql://weever:4333/RIOStaff”, “select * from STAFF”)
        as $e,
      query(“poet://LOCAL/data/SGML_base”,
            “select p.summary.header from Proj p”) as $h
where $e.name = $h.author
```

This example includes two *native* queries: an SQL query to retrieve employees from the `STAFF` database, and an OQL query to retrieve projects headers from the `Proj` database. It then uses a `pick` query (an instruction of our language with syntax and semantics similar to `select` in SQL) to join those two results on the author name. The result is a list of employees who have authored a project.

3.1.1 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* [CACM94], 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 `$d` denotes a project document as shown in Figure 1, then `$d.summary` selects the summary section, and `$d..description` selects all descriptions at any level.[4] These expressions actually return *all* the matching subtrees in a list, unlike other object-oriented approaches where there can only be one possible match (as is the case with OQL for example).

Node types can be used for selection. For example `$d.#ATTR` selects all project attributes (`gate`, `id` and `class`).

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,

`$d.summary.[#FIRST]`	*selects the* `summary` header
`$d.#ATTR[0]`	*get the first attribute (*`gate`*)*
`$d..description[#FIRST]`	*selects the first* `desc`.

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

`$d.#ATTR[#FIRST:#LAST]`	*all attributes (equivalent to* `$d.#ATTR`)
`$d.id&summary`	*the* `id` *and* `summary`
`$d.id:summary`	*the* `id`, `class` *and* `summary`

[4] As descriptions can only occur directly under summaries, the latter is equivalent to `$d.summary.description`.

`$d..header&desc[0]`	*the* `header` *and first* `desc`.

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

3.1.2 Information construction and modification

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

`elem("paragraph", $d..desc)`	*element "paragraph" with descriptions as children*
`attr("version",$d.id)`	*attribute "version" with value equals to* `$d.id`

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). For example, `str($d)` or `str($d..#ELEM)` return a string for the content of the whole project document.

Information can be added, removed or replaced with `adopt`, `excluding`, and `replace`, respectively. For example:

`$d excluding ..desc`	*the project without descriptions*
`$d adopt attr("version",1.0)`	*the project with* `version` *attribute*
`$d replace .gate.#VALUE with 1`	*change the project* `gate` *from 0 to 1*

Other operations include `sort` and `distinct` to sort entries and remove duplicates in a list .

3.1.3 Querying

Native queries can be expressed with the `query` predicate. They take two arguments: the *location* of the data source and the *query* itself. The *location* also identifies the type of the data source and the protocol used to access it. For example we can deduce that the example below is an SQL query, from the protocol "jdbc:msql". The database is located on "weever:4333" in file "RIOStaff".

```
query("jdbc:msql://weever:
4333/RIOStaff",
      "select * from STAFF")
```

select all records from STAFF database

A simpler form of `query` is provided for HTML documents.

`url("http://www.csiro.au/")`	*parses CSIRO home page and return its parse tree*

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

```
pick $p from
query("poet://LOCAL/data/SGML_bas
e", "select project from Proj p")
as $p
where $p..portfolio = "EDC"
```

select project documents that belong to the "EDC" portfolio from the Proj database

3.1.4 Assignments

Assignments are performed with `define`. For example, the variable `$d` used throughout this section could be defined as:

```
define $d as unique query("poet://LOCAL/data/SGML_base",
                          "select p from Proj p where p.id = 'P01'")
```

3.1.5 Miscellaneous

Other features of the language include: access to node types and labels (`type` and `label`), simple arithmetic (`+`, `-`, and comparison operators), simple functions (`empty`, `count`, etc.), boolean operators (`and`, `or`, `not`, etc.), etc.

3.2 Mapping dynamic information into virtual documents

The purpose of the mapping instructions is to convert trees representing dynamic information, into elements of the virtual document. In particular, the mapping instructions must permit these various

SGML constructions: PCDATA, attributes, aggregates, recursive lists. 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 processing instructions that can interact with the static part.

3.2.1 Text

By default, dynamic information is mapped into text. The processing instruction:

```
<?$q>
```

is interpreted as text($q), where text 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).

3.2.2 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 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 an 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).

3.2.3 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.[5] 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>
```

3.2.4 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 query(“jdbc:msql://weever:4333/RIOStaff”,
                         “select * from STAFF”)>
```

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>
```

---

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

Figure 2: Home page example

3.2.5 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>`.

4 A Complete Example

We now illustrate the language through an example application: automatically-generated home pages for members of our group. These pages display a photograph of the employee, with his name, title, contact details, professional interests, and publications (see Figure 2). The employee's details are taken from two databases: a local database that defines attributes specific to our group (like professional interests), and a global database for all employees within our division. Publications are stored in BibTeX format and retrieved through an in-house information retrieval system.

Figure 3 shows the document prescription for this example.[6] The document prescription defines two parameters: the surname and given name of the employee (line 1). Both these parameters must be supplied to generate the virtual document. Lines 2 and 3 retrieve the employee from the local and global databases (TIMStaff and CarltonStaff, respectively). Line 4 includes a photograph of the employee by supplying the SRC attribute to a IMG element. Line 5 outputs the employee's name and title. Line 6 displays the employee's e-mail, which is also defined as a link. If any professional interests are supplied in the database, they are displayed in line 7. Finally, publications, if any, are displayed in line 8.

5 Related Work

Selection of objects in our language is similar to the generalised path expressions found in POQL [ACC97]; however in our case 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

[6] This is actually an excerpt from our Web pages. The original document prescription and the associated virtual document can be seen at: http://star.vic.cmis.csiro.au:8042/RIO/staff.rhtml.

```
1. <?param "sname" as $surname>  <?param "fname" as $firstname>
   <HTML><BODY>
2. <?define $tim as unique
     query("jdbc:msql://weever:4333/TIMStaff",
           str("SELECT Interests from staff
                where Surname=’", $surname,"’"))>
3. <?define $staff as unique
     query("jdbc:msql://weever:4333/CarltonStaff",
           str("SELECT * from staff where Surname=’", $surname,"’"))>
4. <img align="left" hspace="5" width="123">
   <?attributes attr("src",str("http://www/TIM/staff-photos/",
                               $staff.alias,"_small.jpg"))>
5. <?$firstname;$surname><BR> <?$staff.title>
   <P><B>Contact Details</B><BR>
6. <?define $email as str($staff.alias,"@",$staff.domain)>
   E-mail:<a><?attributes attr("href",str("mailto:",$email))>
   <?$email></a><br>
   Telephone: +61 3 9282 2<?$staff.phone><br></P>
7. <?if not empty $tim.interests>
   <P><B>Professional Interests</B><BR>
   <?$tim.interests></P>
8. <?define $pubs as query("pif://star:8042/TIM",$surname)>
   <?if not empty $pubs>
   <?begin>
     <P><B>Publications</B></P>
     <?map $aPub in $pubs>
       <p><?$aPub></p>
   <?end>
   </body></html>
```

Figure 3: A document prescription example

strict typing conventions has been recognised in Lorel [AQM97], which use coercion between types to address the problem. An alternative to path expressions is tree patterns, as in SgmlQL [LMR95]. 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. Various languages for querying the Web in a database style have been developed [AMMT96, KS95, MMM96]. They allow searching for specific elements in pages, as well following links, but they do not support querying the full SGML structure.

DTD conversion has been addressed already by a number of approaches. Some SGML editors can support cut-and-paste between incompatible elements [AQR97]. Tree transformation languages such as TranSID [JKL97] are devoted to DTD conversion. General-purpose languages such as DSSSL, in addition to conversion from one DTD to another, also address presentation issues and conversion to other formats. 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.

Languages for building documents from existing sources of information have been less explored. We have already mentioned PHP [Ler] and w3-msql [Hug]. Other programming approaches include WebMethods [WM97], for application-based extraction of information, and WebL [KM97], a scripting language for Web document processing. Document-based approaches such as Araneous [AMMT96], although primarily aimed at analysing and querying Web pages, also offer limited functionalities for creating new Web pages.

6 Conclusion

The language presented here is one of the several components we think are essential for electronic publishing [PV97]. The language has been implemented and is being tested at our Web site[7], as was

[7] See link at http://www.cmis.csiro.au/TIM/research.html.

shown in section 4. Although shown here with HTML, our approach has been implemented for any SGML document.

Future plans include coupling the language with an editor to facilitate the document authoring process, and refining the document interpretation process to allow parallel interpretation and appropriate treatment of errors (such as when a data source is temporarily unavailable, or its structure has changed, etc.).

References

[ACC97] Serge Abiteboul, Sophie Cluet, Vassilis Christophides, Tova Milo, Guido Moerkotte and Jérôme Siméon, "Querying Documents in Object Databases", *International Journal on Digital Libraries*, 1(1), pp5-19, 1997.

[AQM97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom and Janet L. Wiener, "The Lorel query language for semi-structured data", *Journal of Digital Libraries*, 1(1), pp68-88, 1997.

[ACM97] Serge Abiteboul, Sophie Cluet, and Tova Milo. "Correspondence and translation for heterogeneous data", in *Proceedings of ICDT 97*, Delphia Greece.

[AQR97] E. Akpotsui and V. Quint and C. Roisin, "Type Modelling for Document Transformation in Structured Editing Systems", *Mathematical and Computer Modelling*, 25(4), pp1-19, 1997.

[AMMT96] 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.

[CACM94] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl, "From Structured Documents to Novel Query Facilities", in *Proceedings of the ACM SIGMOD Conference on Management of Data*, Minneapolis, Minnesota, pp313-324, 1994.

[JKL97] Jani Jaakkola and Pekka Kilpeläinen and and Greger Lindén, TranSID: An SGML Tree Transformation Language, University of Helsinki, Department of Computer Science, C-1997-36, May, 1997.

[KM97] Thomas Kistler and Hannes Marais, WebL – A Programming Language for the Web, SRC Technical Note, Digital, 1997-029, December 1997.

[KS95] David Konopnicki and Oded Shmueli, "W3QS: A Query System for the World-Wide Web", in *Very Large Data Bases* (VLDB), Zurich, Switzerland, pp54-65, 1995.

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

[LMR95] Jacques Le Maitre, Elisabeth Murisasco and 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.

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

[Lev93] David M. Levy, "Document reuse and Documents systems", *Electronic Publishing*, 6(4), pp 339-348, December 1993.

[MMM96] Alberto O. Mendelzon, George A. Mihaila and Tova Milo, "Querying the World Wide Web", in *Conference on Parallel and Distributed Information Systems* (PDIS), Miami Beach, Florida, USA, 1996.

[Par97] François Paradis, *The Prescription Language of the RIO Project*, technical report, CMIS, October, 1997.

[PV97] François Paradis and Anne-Marie Vercoustre and Brendan Hills, "A Virtual Document Interpreter for Reuse of Information", to appear in *Electronic Publishing '98*, Saint-Malo, France, 1-3 April, 1998.

[VDH97] Anne-Marie Vercoustre, Jon Dell'Oro and Brendan Hills, "Reuse of Information through Virtual Documents", in *Second Australian Document Computing Symposium*, Melbourne Australia, pp55-64, April 5, 1997.

[VP97] Anne-Marie Vercoustre and François Paradis, "A Descriptive Language for Information Object Reuse through Virtual Documents", in *4th International Conference on Object-Oriented Information Systems* (OOIS'97), Brisbane, Australia, pp299-311, 10-12 November, 1997.

[WM97] WebMethods, *The Web Interface Toolkit*, http://www.webMethods.com/.