A Virtual Document Interpreter for Reuse of Information

François PARADIS[1], Anne-Marie VERCOUSTRE[2], and Brendan HILLS[1]

[1] CSIRO Mathematical and Information Sciences
723 Swanston St, Carlton, VIC 3053, Australia
[2] INRIA Rocquencourt
BP 105, 78153 Le Chesnay Cedex, France

Abstract. The importance of reuse of information is well recognised for electronic publishing. However, it is rarely achieved satisfactorily because of the complexity of the task: integrating different formats, handling updates of information, addressing the document author's need for intuitiveness and simplicity, etc. An approach which addresses these problems is to dynamically generate and update documents through a descriptive definition of *virtual documents*. In this paper we present a document interpreter that allows information to be gathered from multiple sources and combined dynamically to produce a virtual document. Two strengths of our approach are: the generic information objects that we use, which enables access to distributed, heterogeneous data sources; and the interpreter's evaluation strategy, which permits a minimum of re-evaluation of the information objects from the data sources.

1 Introduction

Recent advances in electronic publishing have greatly modified the very concept of *document*. Thanks to direct and almost instantaneous access to information, documents are no longer constrained to be *static*, but can be *dynamic* or *virtual*: ie. they can include pieces of information from other documents or other data sources, transform them, and reflect their updates. This comes as a great help for the document production process, as it largely involves reusing parts of pre-existing documents [10]. Virtual documents do not only reduce the cost of data entry, but also facilitate the validation and maintenance of documents.

Virtual documents have been defined in [8] as *hypermedia documents that are generated on demand, in response to user input*. Such documents have proven to be useful for reuse [6]. There are many ways to implement them on the Web, ranging from simple CGI scripts or Java applets to advanced servers like PHP[1] or w3-mSQL[2]; however all have the disadvantage of imposing a functional or programming-like approach, which makes it difficult for non-experts to write documents. A few representative examples of virtual documents systems are: PEBA[3] [12] which produces natural language

[1] See PHP/FI Home Page at http://www.vex.net/php/.
[2] http://cs1.inf.uni-hohenheim.de/ftp/sw/sun-solaris-2.x/mSQL/w3-msql/2.0/w3-msql.html.
[3] http://www-comp.mpce.mq.edu.au/mri/peba/.

descriptions of animals from a knowledge base, ComMentor [16] which dynamically synthesizes documents from distributed sources to produce personalized content, and DME[4] [8], a question-answer system that generates domain-based explanations. Those systems however are very specific to a domain or application.

Reuse of information is also attracting much attention in the Web community lately, as testified by the "Web" query languages that can retrieve information from semi-structured [9] or structured sources. Examples are: Lorel [1], SgmlQL [11], and [4], later refered to as POQL. These languages will only show their full potential however when integrated to the document production process.[5]

This paper presents a virtual document interpreter which addresses the following problems:

- *Plurality and heterogeneity of sources*. The interpreter is able to query and combine data from various sources and in different formats.
- *Efficient evaluation*. For many applications the generation of the documents is costly but needs to be performed relatively rarely. The interpreter can keep previous results, and, if possible, regenerate documents only when necessary. We push this idea further with our interpreter and allow the storage of intermediate results as well.
- *Integration in the lifecycle of the document*. Writing virtual documents involves formulating queries to data sources, and possibly modifying these queries from the results. This process will be best implemented by coupling the interpreter with an editor, and allowing "partial" evaluation of a document prescription.

The paper is organized as follows. First we give a brief overview of the goals and architecture of our approach. Next we present the data structures used for exchanging information among the data sources. We then go into more detail of the interpreter, describing the structures produced by the parser, and how they are evaluated and stored. Finally, we discuss a possible integration of this strategy with an editor.

2 Virtual documents for electronic publishing

The RIO (Reuse of Information Objects) project aims to develop techniques which can support information reuse in various contexts [19]. The focus of the project is currently on the specification and interpretation of virtual documents to enable reuse of structured information from heterogeneous sources. Figure 1 shows our strategy for virtual document publishing in this context. The instructions for the construction of virtual documents are stored in a *document prescription*, which is processed by the *document interpreter* to generate or update a virtual document. An *editor* facilitates the writing of the document prescriptions; it is connected to the document interpreter in order to provide *dynamic* editing.

The document prescription consists of:

[4] http://WWW-KSL-svc.Stanford.EDU:5915/doc/papers/ksl-96-16.

[5] A move in that direction can already be seen in SgmlQL [11] which can have its queries embedded in SGML documents, and includes some operations to *construct* information.

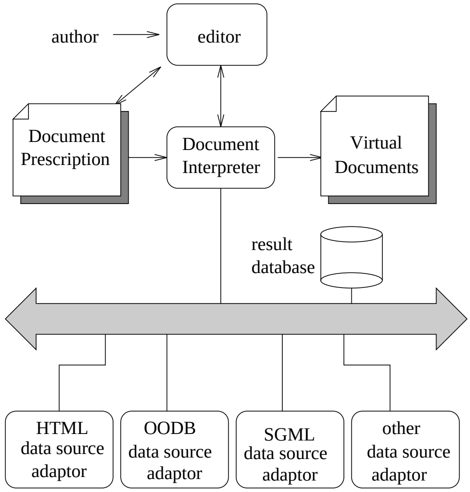

Fig. 1. Integrated approach to virtual document publishing

- *Static data*, or the structure and the text that does not change in the document.
- *Queries*, or the commands needed to generate the dynamic part of the document.
- *Transformation instructions*, to convert the reused information objects into new document objects.

The document prescription is written as an SGML document or as one of its derivatives such as HTML or XML, that might not enforce compliance to a formal DTD. Static data is expressed using normal SGML constructs. Queries and transformation instructions are expressed as SGML *Processing Instructions* (PI).[6] There are two kinds of queries: *native* queries, which send requests to the data sources in their specific language (eg. SQL for a relational database, URL for an HTML server), and *pick* queries, written in an OQL-like language that we designed to combine results and provide search capabilities for semi-structured information.

Figure 2 shows a simple example of a document prescription which generates an HTML document containing a list of staff members. Oviously, this simple example does not show all the features of the language; for a complete and formal description, including examples of joins and combinations of queries, see [20].

The first PI (`pick`) in figure 2 fetches a header from another HTML page, where it appears as the first `<TABLE>` of the `<BODY>` section, and inserts it as SGML code in the document.[7] The second PI (`define`) does not produce any output to the virtual document, but stores the result of an SQL query that retrieves all staff members from a relational database in a variable. The `map` instruction, finally, iterates over the staff

[6] This makes it easy to turn a static document into a virtual document, without having to modify the DTD.

[7] We assume that local references (eg. images on the local servers) will be resolved, ie. converted to global references.

members list, producing an <LI> element for each staff member, with their name as content.

```
<HTML><BODY>
<?pick sgml(body.table[#FIRST]) from
url(‘‘http://www.csiro.au/’’)>
<?define $staff as sql(‘‘select * from STAFF’’)>
<H1>Staff members</H1>
<UL><?map $i in $staff><LI><?$i.name></LI> </UL>
</BODY></HTML>
```

Fig. 2. A simple document prescription

The expressions `body.table[#FIRST]` and `$i.name` in figure 2 are *path expressions*; they perform selections on the results in an OQL-like manner (our syntax is inspired by [4]). A path expression can be seen as the traversal of a tree. An expression `.L` (*dot selection*) finds the children with label `L` one level down the tree, an expression `..L` (*dot-dot selection*) finds the children with label `L` at any level down the tree, and the bracket modifiers (`[R]`) select a particular child or a range of children.

The role of the *document interpreter* is to gather and combine information from the data sources, as instructed in the document prescription, and to map it to the virtual document. The document interpreter must deal with two main problems: the integration of heterogeneous, distributed data (section 3), and the dispatch of queries to data sources (section 4).

3 Information objects

We call *information objects* the pieces of information that the document interpreter gathers from data sources and manipulates in order to generate virtual documents. A common representation of information objects is needed, so that, for example, a relational database table or an HTML document can be treated in the same way by the interpreter. This requirement is quite common for systems that need to fuse various data [13].

Our approach is to define an *interface* to information objects rather than impose a particular data structure. This way no assumption is made as to how the information is stored in the data source. This allows some data sources to build the information objects on demand and thus to avoid unnecessary conversions.

The interface is shown in figure 3 using the IDL language [7]. Each information object has a type, a label, and a list of children.

Information objects have a tree-like structure. A tree node is one of the following:

- an `ELEM` node is a labeled, composite entity, whose children represent structural components. For example, for an SGML element, the label is the tag name, and the children the elements comprised in it.

```
interface Node {
  // *** access
  enum NodeType {ELEM,ATTR,LIST,VALUE};
  NodeType getType();
  string getLabel();
  sequence<Node> getChildren();
  boolean hasDescendant(in string label);
  // *** comparison
  //   return true if the Node matches a label and/or a type
  boolean matchLabel(in string compLabel);
  boolean matchType(in NodeType compType);
  boolean matchLabelType(in string compLabel,in NodeType compType);
  // *** conversion
  string toSGML();
  string toText();
};
```

Fig. 3. IDL interface for the information objects

– a `VALUE` node is some text (stored in the node's label). It cannot have children. In SGML, this corresponds to PCDATA.
– an `ATTR` node gives an attribute for an `ELEM` node. Its label is the name of the attribute, its child (children) gives the value(s) of the attribute (a `VALUE`). This corresponds to the tag attributes in SGML.[8]
– a `LIST` node is an unlabeled, composite entity, whose children represent the list elements. All native queries and path expressions return `LIST`s (possibly empty).

The interface also defines simple comparison instructions (`match`); the document interpreter uses these, in addition to the `getChildren` method, to implement path expressions. Finally, conversion instructions (`toSGML` and `toText`) are used to include the entire information object in a virtual document (see section 4.3).

4 Document interpretation

The interpretation of a document prescription consists of: parsing the document prescription, evaluating and sending queries to data sources, and integrating results in a virtual document. This process is illustrated in figure 4. Central to our approach is the use of a *result database*, which stores intermediate results, allowing the interpreter to re-use previous results if a data source has remained unchanged or is temporarily unavailable.

We now describe each of the interpretation steps in detail.

[8] 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. This gives us a more unified representation for path expressions.

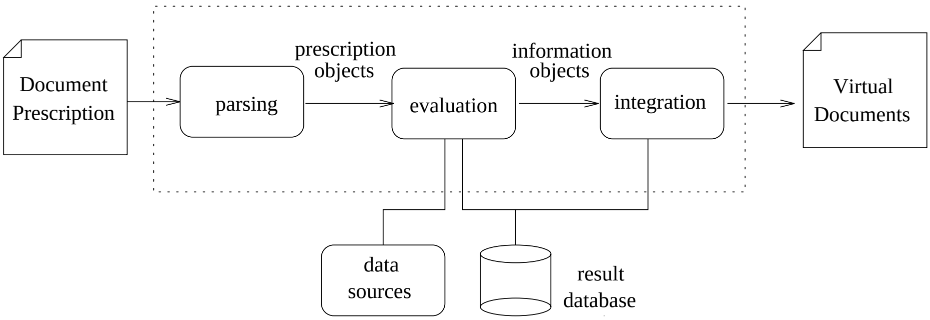

Fig. 4. The document interpretation process

4.1 Parsing

The parsing process translates the document prescription, an SGML file, into a set of *prescription objects*, or objects that can be evaluated by the document interpreter. The aim of this division into prescription objects is to identify the objects that can be stored in the result database, and which other objects they depend upon.

The parser divides the document prescription into prescription objects in the following way:

– Contiguous static data constitutes one prescription object, except when part of a `map`.
– A `map` instruction along with its associated SGML element constitute one prescription object.
– A `define` instruction is split into two prescription objects: one for the variable and one for the value assigned to it.
– A top-level `pick` instruction or path selection is a prescription object.
– Native queries are prescription objects.
– Variables are prescription objects, except for those declared in the `from` clause of a `pick`, or as the iterator variable of a `map`.

For example, the document prescription shown in figure 2 would generate the following prescription objects:

```
A.  <HTML><BODY>
B.  pick sgml(body.table[#FIRST]) from url(‘‘http://www.csiro.au/’’)
C.  url(‘‘http://www.csiro.au/’’)
D.  $staff
E.  sql(‘‘select * from STAFF’’)
F.  <H1>Staff members</H1><UL>
G.  <?map $i in $staff><LI><?$i.name></LI>
H.  </UL></BODY></HTML>
```

One of the roles of prescription objects is to capture the dependencies between the dynamic parts of the document. Let $dep(\alpha, \beta)$ be a binary function that returns $true$ if

α has a dependency with β, and $false$ otherwise. In practice this means that whenever β changes, α needs to be updated as well. In our example, we have:

$$dep(B, C) = true, dep(D, E) = true, dep(G, D) = true$$

The set of dependencies for a prescription object α is given by:

$$D_\alpha = \{\beta | dep(\alpha, \beta) = true\}$$

The function dep introduces a partitioning of the prescription objects that we call the *prescription trees*. In our example, there are 6 prescription trees, with the following root nodes: A, B (with child C), D (with child E), F, G (with child D), and H.

4.2 Evaluation

The aim of the evaluation process is to produce information objects from prescription objects. It involves sending native queries to data sources, performing more selection and transformation[9] on these results (path expressions and `pick` queries), assignments (`defines`), repetition of tags (`map`), etc. However this evaluation only occurs if the prescription objects actually need updating from the previous evaluation. In this section we explain how evaluation occurs using information from the parsing and from the result database.

The result database stores the prescription objects, along with information about when they and their dependencies were last generated. The generation time of a prescription object α is given by τ_α: it is equal to the current time at generation. The generation time of a dependency α to β is given by $\tau_{\alpha,\beta}$: it is equal to the generation time of prescription object β. If α does not have a dependency with β, then $\tau_{\alpha,\beta}$ equals $\perp$, with, for any time t, $\perp < t$. The set of stored dependencies for a prescription object α is given by:

$$SD_\alpha = \{\beta | \tau_{\alpha,\beta} > \perp\}$$

Prescription objects are evaluated in the post-order traversal of the prescription trees, ie. the children are evaluated first, and the dependents or roots of the trees are evaluated last. In general, a prescription object α does not need to be updated and can be reused from the result database if the two following conditions are met:

$$D_\alpha = SD_\alpha$$
$$\forall \beta \tau_{\alpha,\beta} = \tau_\beta$$

that is, if the dependencies of the parsed object and of the stored object are the same, and if the generation times of the stored dependencies are up-to-date. Note that a prescription object gets updated even in the event that one of the dependencies has "gone back" in time.

For the evaluation of a native query encoded as prescription object α, the document interpreter must be able to compare τ_α with the revision time of the corresponding data

[9] Although not described in this paper, our language allows the user to construct new information objects, to add or remove information objects, etc.

source, and send the query to the data source only if they differ. This is quite easy to achieve with databases, but may present some problems for other data sources[10]; if the data source lack this capability, then it returns the current time.

Consider the first evaluation of our example document, at time t_1. Supposing that the URL and the database were last updated at t_0, the following updates would occur (shown here in their evaluation order):

$$\begin{aligned}
\tau_A &= t_1 \\
\tau_C &= t_0, \tau_{B,C} = t_0, \tau_B = t_1 \\
\tau_E &= t_0, \tau_{D,E} = t_0, \tau_D = t_1 \\
\tau_F &= t_1 \\
\tau_{G,D} &= t_1, \tau_G = t_1 \\
\tau_H &= t_1
\end{aligned}$$

As this was the first evaluation, all objects need to be updated[11], and will be stored in the result database. This also includes static objects A, F, and H, even if the evaluation in this case only consists of returning a string corresponding to the prescription object.

Suppose now that the staff database changes at time t_2, and the document is re-evaluated at time t_3. Then the necessary updates are:

$$\begin{aligned}
\tau_E &= t_2, \tau_D = t_3, \tau_{D,E} = t_2 \\
\tau_G &= t_3
\end{aligned}$$

This simple technique guarantees minimal re-evaluation in case of a modification of the document prescription. Suppose our document prescription is altered so that the staff list is retrieved from database STAFF-CMIS instead of STAFF. The parsing would be similar except that object D would have a new dependent, E', corresponding to `sql(select * from STAFF-CMIS)`. Upon re-evaluation of the document, the update of D would be enforced, because the set D_D ($\{E\}$) is different from SD_D ($\{E'\}$). E might still be kept in the database for version control purposes.

This processing also ensures a correct evaluation in case of multiple assignments to a variable in a document, although this practice is not recommended as it defeats the purpose of the result database.

4.3 Integration

This last step is responsible for combining and mapping results of the evaluation process into virtual documents. As those results are information objects, it is straightforward to combine and include them in a document. An information object can be mapped to the virtual document in two ways:

---

[10] For HTML pages, the HTTP protocol allows access to the modification date of a file, but this does not guarantee that its *content* is not newer: it could be generated or include other files (eg. images).

[11] It is assumed that before the first evaluation all generation times are equal to $\perp$.

- As text, to get the textual content of the answer. The `VALUE` nodes of the information object (except the children of `ATTR` nodes) are concatenated (with whitespaces to separate them).
- As SGML, to get the structure of the answer. The `ELEM` node is the tag name, with its `ATTR` children for attributes, and other nodes as content. `VALUE` nodes are PCDATA.

By default results are included as text, unless specified otherwise with the function `sgml`. The function `sgml` performs a direct mapping of the information objects into SGML; however we envisage for the future more sophisticated DTD mapping and conversions.[12]

5 Coupling with an editor

An important strength of our virtual document system is that its focus is on documents rather than processing because the virtual document prescription is itself a document. This brings two advantages: the prescription can be stored, archived, indexed and searched like any other document; and more importantly the user can conceptualise and interact with the document just like any other document. We feel that this second point is in keeping with the aim of our virtual document system which is to make it easier for people to write virtual documents. A crucial part of our virtual document system is an editor which makes it easy to write virtual document prescriptions.

As described in section 2, our virtual document prescription language has three parts: static data, queries to external data sources, and instructions for transforming the result of queries. There are different requirements for editing these three parts of a document prescription.

5.1 Editing static data

Since our virtual document prescription language is based on the structured document language SGML, an editor for the static part of the virtual document prescription has the same requirements as a structured document editor. The most important of these is to provide the author with both a view of the document structure and a representation of the document appearance.[13] The editor should enforce the document's structure so that the result still conforms to the structural requirements (eg. the DTD in the case of SGML) but should be flexible enough to allow the user to "break" the structure temporarily whilst editing . Our interpreter does not actually check that the virtual document prescription conforms to the required structure, rather it relies on the editor to do that.

[12] This mapping to a DTD can be achieved at the moment using the construction and modification instructions of our language.

[13] Since one of the ideas of SGML is that different parts of the document may be used for different purposes [18, p.10], a "WYSIWYG" editor for an SGML document is quite different to a traditional WYSIWYG word processor.

5.2 Editing data queries

In order to enable the author to construct and edit data queries in a virtual document prescription the editor must provide a view of the data sources that those queries will access. To edit a query, the author must be able see both the structure of the data source (eg. the schema of a relational database, the DTD of a SGML document repository, the class structure of an OODB) and the data that is available from the source. The editor should present different kinds of data with different browsing paradigms as it is important that the presentation of the structure accurately models the data source's native query language so that the author can correctly conceptualise the way that the data source query language works.

- Relational databases: tables displayed on a graph with relations indicated by edges.
- OODB: class structure diagram as used in common OO methodologies such as Booch [3] or Rumbaugh [17].
- SGML: either a tree structure of the DTD structure or a form-based approach as used in InContext or Grif [14].

5.3 Editing transformation instructions

The editor must assist the author to edit the virtual document prescription's transformation instructions. This requires the author to be able to visualise our internal tree structure (as described in section 3) as it is this structure which is processed by the transformation instructions.

5.4 Virtual document preview

The virtual document prescription editor must provide a facility for the author to preview the virtual document as it will be instantiated. We aim to provide the user with the option of a view which displays the structure and contents together, or with two separate windows – one showing the virtual document prescription, and the other showing the instantiated virtual document – similar to structured editors that display the document structure and presentation in different windows. This requires the editor to communicate with the virtual document prescription interpreter so that either the whole document, or portions of it, may be interpreted and sent to a "presentation" module. The preview facility must not only be able to resolve data source queries and transformation instructions, but it must also be able to display error and boundary conditions of the results of queries and instructions. For example, what will the virtual document look like if a query returns an error, no data or far more data than was expected?

A requirement of this preview facility is that it is able to perform partial evaluation of the document. By this we mean that as the user edits queries in a part of the document prescription, the editor should be able to display the changes in that part without completely re-evaluating the document. Since it is possible to have quite complex dependencies within a virtual document prescription, we are developing a partial evaluation strategy which aims to allow the user to control how much of a document is evaluated when a change is made. As an example, while a part of a document prescription is under development it will be re-evaluated each time it is changed and it may therefore become inconsistent with other parts of the document that depend on it. It must be possible for the user to request a re-evaluation of the parts of the document prescription which depend on the changed part to remove these inconsistencies once it becomes stable.

6 Conclusion

We have described a virtual document approach for electronic document publishing that facilitates reuse of information. Our view of virtual documents is a somewhat restricted form of active documents [15]. The focus in this paper has been on the document interpretation process. This point needs greater understanding for the development of virtual document systems.

A natural addition to the document interpreter would be to allow non-sequential evaluation of documents. Two prescription objects that do not share any dependents, and for which the data sources are "well-behaved" (ie. have no side-effects on one another) could be evaluated in parallel. Given the distributed nature of the data sources, the interpretation would greatly benefit from this parallelization. Another interesting addition would be to provide version control, which should be quite straightforward as we already store the dependencies and time-stamps.

The approach is currently being implemented and tested on a set of dynamic Web pages[14]. We have not implemented any of the optimisations proposed in [5] for our query language; we do not feel this will have a major impact on our approach as the resource discovery is done by the data source, and our query language acts as a selection on the results. We are also working on an integration with the Thot editor [2] that will satisfy the requirements defined in section 5. This integration will add some functions that are important for the editing of virtual documents: it will permit a *partial* evaluation of the document prescription, so that only the part of the document that is currently being edited needs to be evaluated, and allow *incremental* evaluation so that even parts that are incomplete in the document prescription can be evaluated.

References

1. 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):68–88, 1997.
2. Stéphane Bonhomme, Vincent Quint, Hélène Richy, Cécile Roisin, and Irène Vatton. *The Thot User's Manual*, 1997.
3. Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, Ca., 1994.
4. 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*, pages 313–324, 1994.
5. Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montréal, Québec, Canada*, pages 413–422, June 4–6 1996.

[14] See our Web site at http://www.cmis.csiro.au/TIM/.

6. Franca Garzotto, Luca Mainetti, and Paolo Paolini. Information reuse in hypermedia applications. In *Seventh ACM Conference on Hypertext, Washington, DC, USA*, March 16–20 1996.
7. Object Management Group. OMG-IDL syntax and semantics. In *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, chapter 3. July 1995.
8. Thomas R. Gruber, Sunil Vemuri, and James Rice. Model-based virtual document generation. Technical Report KSL-96-16, Knowledge Systems Laboratory, Stanford University, May 1996.
9. Joachim Hammer, Hector Garcia-Molina, J. Cho, Arturo Crespo, and Rohan F.M. Aranha. Extracting semistructured information from the web. In *Proceedings of the Workshop on Management of Semistructured Data, Tucson, Arizona*, pages 18–25, May 1997.
10. David M. Levy. Document reuse and document systems. *Electronic Publishing*, 6(4):339–348, December 1993.
11. Jacques Le Maitre, Elisabeth Murisasco, and M. Rolbert. From annotated corpora to databases : the SgmlQL language. In *CSLI lecture notes, collection linguistic databases.* Cambridge University Press, to appear.
12. Maria Milosavljevic, Adrian Tulloch, and Robert Dale. Text generation in a dynamic hypertext environment. In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia*, January 31 – February 2 1996.
13. Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB), Bombay, India*, pages 413–424, 1996.
14. Vincent Quint and Irène Vatton. Grif: an interactive system for structured document manipulation. In *Text Processing and Document Manipulation, Proceedings of the International Conference, J. C. van Vliet, ed., Cambridge University Press*, pages 200–213, 1986.
15. Vincent Quint and Irène Vatton. Making structured documents active. *Electronic Publishing - Organization, Dissemnation and Design*, 7(2):55–74, June 1994.
16. Martin Röscheisen, Christian Mogensen, and Terry Winograd. Shared web annotations as a platform for third-party value-added information providers: Architecture, protocols, and usage examples. Stanford Integrated Digital Library Project STAN-CS-TR-97-1582, Computer Science Dept., Stanford University, November 1994.
17. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
18. Eric van Herwijnen. *Practical SGML: Second Edition*. Kluwer Academic Publishers, 1994.
19. Anne-Marie Vercoustre, Jon Dell'Oro, and Brendan Hills. Reuse of information through virtual documents. In *Second Australian Document Computing Symposium, Melbourne Australia*, pages 55–64, April 5 1997.
20. 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*, pages 299–311, 10–12 November 1997.