January 1999

WysiWyg Web Wrapper Factory (W4F)

Arnaud Sahuguet
University of Pennsylvania

Fabien Azavant
École Nationale Supérieure des Télécommunications

Working Paper, 1999, 22 pages. Unpublished.

WysiWyg Web Wrapper Factory (W4F)

Abstract
In this paper, we present the W4F toolkit for the generation of wrappers for Web sources. W4F consists of a retrieval language to identify Web sources, a declarative extraction language (the HTML Extraction Language) to express robust extraction rules and a mapping interface to export the extracted information into some user-defined data-structures. To assist the user and make the creation of wrappers rapid and easy, the toolkit offers some wysiwyg support via some wizards. Together, they permit the fast and semi-automatic generation of ready-to-go wrappers provided as Java classes. W4F has been successfully used to generate wrappers for database systems and software agents, making the content of Web sources easily accessible to any kind of application.

Keywords
Web wrapper, information extraction, HTML parsing, HTML to XML conversion

Comments
Working Paper, 1999, 22 pages. Unpublished.

WysiWyg Web Wrapper Factory (W4F)

Arnaud Sahuguet
Department of Computer and Information Science
University of Pennsylvania
sahuguet@saul.cis.upenn.edu

Fabien Azavant
École Nationale Supérieure des Télécommunications
Paris, France
fabien.azavant@enst.fr

Abstract

In this paper, we present the W4F toolkit for the generation of wrappers for Web sources. W4F consists of a retrieval language to identify Web sources, a declarative extraction language (the HTML Extraction Language) to express robust extraction rules and a mapping interface to export the extracted information into some user-defined data-structures. To assist the user and make the creation of wrappers rapid and easy, the toolkit offers some wysiwyg support via some wizards. Together, they permit the fast and semi-automatic generation of ready-to-go wrappers provided as Java classes. W4F has been successfully used to generate wrappers for database systems and software agents, making the content of Web sources easily accessible to any kind of application.

Keywords:
Web wrapper, information extraction, HTML parsing, HTML to XML conversion.

1 A need for Web wrappers and design guidelines

The Web has become a major conduit to information repositories of all kinds. Because it is based on open standards, it has entry costs for publishers and offers free navigation tools for end-users, it has become the de-facto standard to publish information.

In most cases, the access to information is granted through a Web gateway with forms as a query language and HTML as a display vehicle. If this architecture offers a convenient access to human users, it is not suitable for computer programs.

An unfortunate consequence is that Web information sources exist independently of one another, like isolated information islands. For instance, when you go to your favorite on-line bookstore, should you want to get a price quote in French Francs and not in US Dollars, you would have to (1) get the price from the bookstore and (2) go to another Web service that offers currency exchange rate to do the conversion. And you would have to do it *by hand* by clicking, fetching and reading Web documents.

What we would really like is automation of the entire process, Web-awareness among Web services (services taking advantage of one another) and interoperability (between Web sources and legacy databases or among Web sources themselves).

To reach these goals, there is a strong need for Web *wrappers*. In the database community, a wrapper is a software component that converts data and queries from one model to another. In the Web environment, its purpose should be to convert information implicitly stored as an HTML document into information explicitly stored as a data-structure for further processing.

Its role is actually three-fold: (1) retrieving the Web document, (2) extracting information from it and (3) exporting it in a structured way. Moreover, the wrapper should be able to cope with the volatile nature of the Web environment: network failures, ill-formed documents, change in the layout, etc.

Unfortunately, most Web wrappers today are written in a very ad-hoc way, with the semantics (if any) hidden inside the code itself, which makes it hard for maintenance and to reuse. In order to build robust wrappers, we identify some key issues concerning their design.

■ **Modular layered architecture:** The design should acknowledge the 3 separate layers – retrieval, extraction, mapping – in order to offer independence and reuse: a layer could be identical for different Web sources (if for instance they use the same query form).

For the wrapper, the information flow among the layers can be described as follows. A to-be-processed HTML document is first retrieved from the Web according to one or more **retrieval rules**.
Then **extraction rules** are applied on the retrieved document to extract information. Finally, this information is mapped into structures exported by the wrapper to some upper-level application, according to **mapping rules**.

It is important to note that the extraction layer is (on purpose) not capable of doing data manipulation (à la XML-QL [5] for instance): data manipulation is now the responsibility of the upper-level application. The rationale here is to keep every layer as simple as possible by splitting responsibilities. It also permits to have some reusable layers to describe information sources.

■ **Declarative specification:** The specification of the each layer should be declarative in order to make it implementation independent, understandable, maintainable and reusable. Declarative specifications also permit clear semantics. Another big advantage is that such specifications can be exchanged and imitated, in the same way people copy and paste Web page fragments by looking at the HTML source or use templates.

■ **Multi-granularity extraction:** It is also crucial to recognize that the information contained in a Web document might present different granularities. Navigating the HTML tag hierarchy can be useful to handle the global *inter-tag* organization of the document (sections, lists, tables, etc.).
But sometimes, it is crucial to reach *intra-tag* information like a comma-separated enumeration inside a table cell. Therefore finer grain extraction tools like regular expressions are necessary. Both types of extraction are complementary and necessary to capture the meaning of a Web document.

■ **Precise semantics:** Using clearly defined object-models is a requirement for declarative specifications. An abstract representation of the HTML document makes document navigation easier and more robust. Having an object-model for the information extracted from the document is also important in order to offer some expressive tailored mappings.

■ **Ease of use and reuse:** The generation of the wrapper as well as its

future utilization and maintenance should be made easy through some support tools (like wizards) that guide the user during the various stages of the creation of wrappers. To make Web resources largely available, the design of Web wrappers must not be restricted to experts.

In the rest of this paper, we present the W4F toolkit that follows the design guidelines mentioned above and aims at helping users create wrappers for Web sources. The various components of the toolkit and the their interaction within the information flow are presented in Figure 1.

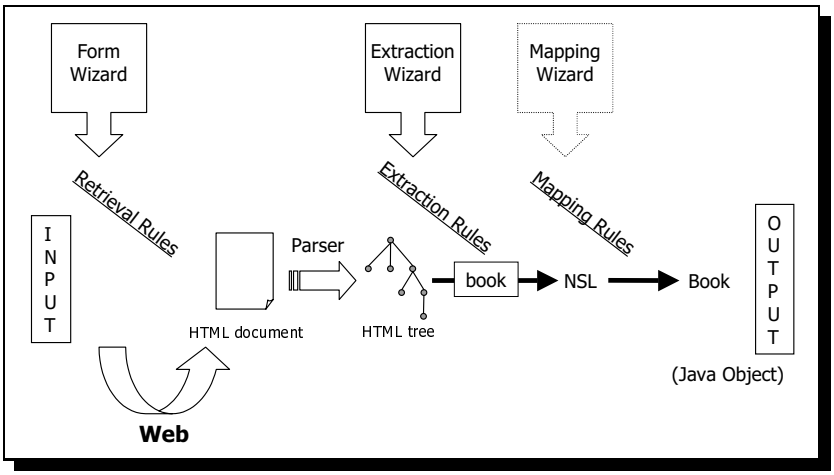

Figure 1: The W4F toolkit

An HTML document is first retrieved from the Web according to one or more retrieval rules. Once retrieved, it is fed to an HTML parser that constructs a parse tree following the Document Object Model [17].
Extraction rules are then applied on the parse tree and the extracted information is stored in our internal format (NSL). Finally, NSL structures are exported to the upper-level application, according to mapping rules.
Rules that determine the process are generated by the user with the help of the wizards. Rules are compiled using the W4F compiler into Java classes that can be run as a stand-alone application using the W4F runtime.

Along this paper, examples are motivated by the building of a Web agent that helps a user when purchasing books on-line. The role of the agent is to gather information about a given book (or set of books) and return a descrip-

tion with a price expressed in the user's own currency. Book information is extracted from Amazon.com. Currency conversion rates come from Yahoo!. For both data sources, we have to build a Web wrapper. The agent's output is an XML document.
Our purpose here is to demonstrate, using our wrapper framework, automation, interoperability between Web sources and conversion of HTML into XML.

The rest of the paper is organized as follows. Section 2, 3 and 4 detail the 3 wrapper layers. Section 5 present the various components of the toolkit involved in the wrapper creation and their role during information processing. In Section 6, we use our freshly brewed wrappers to build the Web agent. We present some related work in Section 7 before we conclude.

2 Retrieving Web documents

The initial job of the Web wrapper is to retrieve the to-be-processed Web document. The retrieval layer is in charge of issuing an HTTP request to a remote server and fetching the corresponding HTML page as any Web browser would do. For the wrapper, it is completely transparent, all the job being done in the background by the retrieval layer: creation of the HTTP request, management of connections, handling of redirections and authorizations, etc.

The retrieval layer is described by a set of retrieval rules that look like an interface definition: the name of the rule is followed by the list of parameters it takes, the type of the method (GET or POST) and the corresponding url (see Figure 2).

The url might contain some variables to be replaced by their string value in order to offer parameterization.
For the POST method, parameters can be specified using the `PARAM` keyword. Some other information to be included in the HTTP request can also be specified.

Some retrieval rules for the two data-sources of our bookstore example are presented in Figure 2.

```
// Amazon.com
RETRIEVAL_RULES ::
get(String keyword)
 {
 METHOD: POST ;
 URL: "http:/www.amazon.com/exec/obidos/xxxx";
 PARAM: "keyword-query" = $keyword$, "mode" = "books";
 }

// Yahoo! (currency conversion)
RETRIEVAL RULES ::
   get()
   {
   METHOD: GET;
   URL:    "http://quote.yahoo.com/m3?u";
   }
```

Figure 2: Retrieval layer specifications for our two wrappers

3 Extracting information using HEL

Once the document has been retrieved, information has to be extracted from it. A wrapper can have an arbitrary number of extraction rules for a given document. As mentioned above, extraction rules are expressed using the HTML Extraction Language (HEL). In this section, we offer an *aperçu* of the language. The full details can be found in [14].

Building the HTML parse tree

The extraction is not performed on the document itself (a document is just a big string) but on an abstract representation of it, better suited for high-level manipulation. Thus each Web document is first transformed into a parse tree corresponding to its HTML hierarchy according to the *Document Object Model* [17].

Such a tree consists of a root, some internal nodes and some leaves. Each node corresponds to an HTML tag (text chunks corresponds to PCDATA nodes). A leaf can be either a PCDATA or a *bachelor* tag[1]. Given this, it is important to note that there is a 1-to-1 mapping between a valid HTML

[1] A bachelor tag is a tag that does not require a closing tag, like `<IMG>` or `<BR>`.

document and its tree. Non-leaf nodes have children that can be accessed by their label (the label of the HTML tag) and their index (the order of appearance).

Two ways to navigate the tree

Navigation along the abstract tree is performed using path-expressions [1].

The first way is to navigate along the **document hierarchy** with the `"."` operator.
The path `'html.head[0].title[0]'` will lead to the node corresponding to the `<TITLE>` tag, inside the `<HEAD>` tag. This type of navigation offers a canonical way to reach each information token (to each chunk of text surrounded by a tag corresponds a unique hierarchical path) but is also limited.

The second way is to navigate along the **flow of the document**, with the `"->"` operator. It really corresponds to a navigation of the document that follows its reading or display order.
More formally, the path `'html->table[0]'` will lead to the first `<TABLE>` tag found in the depth-first traversal of the abstract tree starting from the `<HTML>` tag. The interest of the operator is that the navigation can for instance jump from one leaf of the tree to another node, down on the document flow. This operator increases considerably the expressivity of HEL, since it permits to cope with irregularities of structure. It is also useful to create navigation shortcuts.

Both operators apply to an internal node of the tree and return one (or more) child according to a label name (e.g. `html`, `title`, etc.) and an index value. Index ranges can be used to return arrays of nodes, like `[1,2,3]`, `[7-]` or the wild-card `[*]`. When there is no ambiguity, the index value can be omitted and is assumed to be zero. As an illustration, `'dt[*]'` in Figure 3 is used to return the list of book entries.

Extracting node information

Extraction rules are not concerned by nodes themselves but by the information they carry. From a tree node, we can extract its text value `".txt"`. The text content of a leaf is empty for a bachelor tag and corresponds to the chunk of text for PCDATA. For internal nodes, the text value corresponds to

the recursive concatenation of the sub-nodes, in a depth-first traversal. The underlying HTML source is extracted using `".src"`.

Some properties of the node like the value of some attributes or the number of children can be retrieved using `"getAttr"` and `"numberOf"`. In Figure 3, we extract the hyperlink leading to the book information using `.b[0].a[0].getAttr(href)`.

```
EXTRACTION_RULES ::
books = html.body.table[2].tr[0].td[1].ul[0].li[2].dl[0].dt[*]
 ( .b[0].a[0].pcdata[0].txt                                  // title
#  .b[0].a[0].getAttr(href)                                  // url
# ->dd[0].pcdata[0].txt, match /Published (19[0-9]{2})/     // year
# ->dd[0].pcdata[0].txt, match /(.*?)\//, split /, /        // authors
# ->dd[0].pcdata[1].txt, match /(\$[^ ]+)/                  // price
  );
```

Figure 3: Amazon.com document and its extraction rule.

Using the power of regular expressions

The relevant information might not be entirely captured by the HTML structure available through the document object-model (e.g. an enumeration inside a table cell): that's where regular expression patterns can be useful.

HEL provides two operators `match` and `split` that follow the Perl syntax (see [18]).

The `match` operator takes a string and a pattern, and returns the result of the matching. Depending on the nature of the pattern[2], the result can be a string or a list of strings.

[2]The number of parenthesized sub-patterns indicates the number of items returned by the match. In the example (see Figure 3), the `match` will return a pair (title, year).

The `split` operator takes a string and a separator as inputs and returns a list of substrings.

These operators can be used in cascade: the operator is applied to each element of the previous result.
In Figure 3, we use the `match` operator to identify the list of authors (everything before the first `"/"`) then apply a `split` to extract each author from the list where `","` is used as a separator.

Enforcing constraints

```
SCHEMA ::
float value;

EXTRACTION_RULES ::
value = html.body.center.table[2].tr[i].td[j].txt
WHERE   html.body.center.table[2].tr[0].th[j].txt =~ "FF"
AND     html.body.center.table[2].tr[i].th[0].txt =~ "U.S.";
```

Figure 4: Yahoo! document and its extraction and mapping rules rule.

As mentioned above, array elements can be specified using wild-cards or index values. They can also be defined using variables to which conditions can be attached by introducing a `WHERE` clause. Conditions cannot involve nodes themselves but only their properties and various comparison operators are offered by the language.

As an illustration of the expressive power of constraints, in Figure 4 we extract the content of a table cell identified by its column and row entries. We are looking for the cell that corresponds to row `'US'` and column `'FF'`, in order to extract the exchange rate between US$ and French Francs.

Creating nested structures

For many wrappers, extraction means extracting *one* single piece of information (i.e. one string) from the document. We really think that extraction has to capture as much structure as it can from the document. If the information we are interested in is a list of books, we should be able to extract a list. If a book is composed of various pieces (title, price, etc.), the extraction should be able to capture them altogether, because they really come together.

Therefore our language provides the fork operator `"#"` to construct nested string lists (NSLs) by following multiple sub-paths at the same time. It permits to extract structures and not only single information tokens. This is particularly useful when information spread across the page need to be put together. For a book (see Figure 3), we put together the `title`, the url, the `year`, the `price` and the list of `authors`.

The fork operator can be used more than once inside a path expression and forked paths can be of any complexity. The fork operator is different from the other operators because it permits to build irregular structures where elements of a list can have different depth. It is up to the mapping layer to know how to deal with this irregular nested structures.

4 Mapping information

The information extracted by the evaluation of one extraction rule is stored in our NSL internal format. The purpose of NSL is to store strings (every piece of information extracted is regarded as a string) but within some arbitrarily complex structures. The NSL datatype can formally be defined by: $NSL = null + string + listof(NSL)$.

The structure of the NSL (levels of nesting, etc.) is fully defined by the rule itself. The use of an array or a split in the rule will result in one extra level of nesting for the result. For the Yahoo! wrapper, the NSL result will be a string. For the Amazon wrapper, it will be a list of items (because of the `'dt[*]'`), each item being itself a list of 5 items (because of the repeated use of the `'#'` operator), the fourth being a list of strings (one string per author, because of the `split`).

The philosophy of W4F is to have wrappers return NSLs that can be freely consumed by some upper-level applications. When there is no specified

```
SCHEMA ::
Book[] books

public class Book
{
   String title; int year; float price; String[] authors;
...
   public Book(NestedStringList nsl) {
      title   = (String)   nsl[0];
      year    = (int)      nsl[1];
      authors = (String[]) nsl[2];
      price   = (float)    nsl[3];
    }
...
```

Figure 5: Mapping layer specification for the Amazon wrapper.

mapping, the extracted information is returned as an NSL. W4F knows how to convert NSLs into Java base types (string, int, float) and their array extensions. For instance, the value extracted from Figure 4 is automatically converted to a float. This default mapping can only be used for simple NSL and regular structures.

For more complex structures, the user can also define his own mappings. In the current implementation, it means providing a Java class with some suitable constructors that know how to consume the NSL and produce a valid instance of the corresponding class.

When extracting our book entries, we want to capture a lot of information concerning a given book. We also want to be able to manipulate the extracted information as a book object with book-related methods. In this case, we want the wrapper to return a `Book` and we have to provide the corresponding Java class. A possible mapping for Amazon.com database wrapper is presented in Figure 5.

As of today, W4F offers non-declarative user-defined mappings via some Java classes. It also offers a declarative mapping to the K2 object-model [4] and a similar one for XML is on its way.

5 The toolkit

Part of the role of the toolkit is to support the user with efficient tools to ease and speed-up the creation of each layer, specially the design of extraction rules that can require a thorough understanding of the underlying HTML document.
In this section we present the various elements of the W4F toolkit.

The role of the **form-wizard** is to assist the user in writing retrieval rules against Web forms. It allows the user to capture the various components of an HTML form by simply clicking on the corresponding submit button. The wizard provides information such as: url of the CGI program, method type and the list of inputs with their name, type and default value. Figure 6 shows the information returned to the user for the bookstore example: this is exactly what we need in order to write the retrieval layer! (compare with Figure 2)

Figure 6: The **form-wizard** in action.

The form-wizard is implemented with a CGI script that takes the Web page where the form is and returns annotated with some JavaScript extensions (see Appendix 9.1 for some implementation details).

The **extraction-wizard** assists the user in writing extraction rules against

Web documents. For a given Web document, the user is presented the same document annotated: it appears exactly as the original one (compare figures 3 and 7). The user simply clicks on the piece of information he is interested in and the wizard returns a corresponding extraction rule. Like the form-wizard, it is implemented by annotating the original document. The details of the annotation are presented in Appendix 9.2.

It is important to note that our wizard returns a canonical path for *the* piece of information clicked on by the user. The wizard cannot deal with collections of items. Moreover, the path is expressed in the HEL language using only the "." operator: no constraints nor regular expressions. The returned extraction rule is not always the best one (in terms of robustness or genericity) but it is a good start anyway: compare the path returned by the wizard (Figure 7) with the one effectively used inside the wrapper (Figure 2).

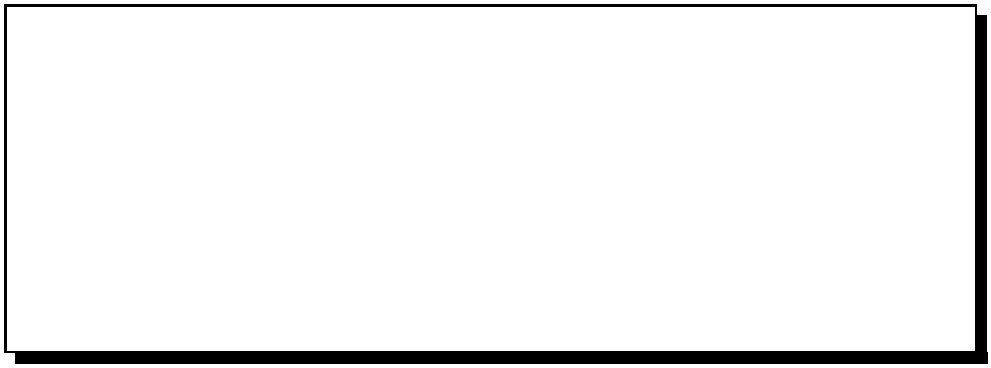

Figure 7: The **extraction-wizard** in action.

The **mapping-wizard** helps the user create mapping from NSLs to user-defined data-structured. As of today, it has not been implemented.

The **W4F parser** is a HTML-3.2 compliant parser. Given the permissiveness of HTML and the way people are writing it, the parser has to recover from badly-formed documents.

The **W4F compiler**'s role is to generate a Java class for a Web wrapper. Given the declarative description of the wrapper (as presented in Figures 2, 3 and 8), the compiler generates a Java class file that can be called from any Java program.

The **W4F runtime** permits to use the generated wrappers as stand-alone applications. The run-time is the set of Java classes necessary to run stand-alone application. The package (jar file) is less than 200kb and is suitable for embedded applications like mobile agents or applets.

6 Putting it all together

```
    Book[] books = Amazon.get(args[0]).books;   // list of books
    float coeff  = Yahoo.get();                 // conversion coeff
    for(int i=0; i<books.length; i++)
      System.out.println( books[i].toXML("FFR", coef) );
```

Figure 8: Pieces of Java code for the bookstore application.

First the two wrappers (see Figures 3, 4 and 5) are compiled into two Java classes: `Amazon.class` and `Yahoo.class`. Now that we have properly defined our wrappers for both sources, we can focus on the upper-level application. Our main program takes an author name or a book title as an input and returns a list of matches using the Amazon wrapper. We use the Yahoo! wrapper to extract the currency exchange rate. For each book, the Amazon wrapper permits to extract the title, year, authors and price of each item of the list. The price is converted into French Francs before the application outputs the result as XML. The Java code for the core of the application as well as its output are presented in Figures 8 and 9.

```
<?XML version="1.0"?>
<BOOK> <TITLE>A Darkness at Sethanon</TITLE>
       <AUTHOR>Raymond E. Feist</AUTHOR>
       <PRICE CURRENCY="FFR">31.35</PRICE> </BOOK>
<BOOK> <TITLE>Daughter of the Empire</TITLE>
       <AUTHOR>Raymond E. Feist</AUTHOR><AUTHOR>Janny Wurts</AUTHOR>
       <PRICE CURRENCY="FFR">29.15</PRICE> </BOOK>
  ...
```

Figure 9: Output of the main application.

7 Related work

In this section, we compare our approach to others, with respects to the various key aspects of the generation of Web wrappers.

Retrieval layer

Some frameworks like WebL [8] and WIDL [2] offer some advanced features for the retrieval. In WIDL, Web sources are described declaratively in term of services, including recovery from failure with retries and alternate retrieval. In WebL (which is a general purpose programming language for the Web), the retrieval consists of writing code using some high-level methods provided by the language. An interesting point is that the semantics of retrieval can be specified using some language constructs (*service combinators*).

In W4F, the retrieval is described declaratively. Issues like recovery or the exact semantics of the retrieval are not addressed[3], in order to keep the layer as simple as possible.

Extraction layer

The issue related to extraction is two-fold: what document structure to use for extraction and how to express it.

On the one hand, a Web document can be viewed as a flow of tokens and the processing requires regular expressions (Tsimmis [7]) or expressive grammars (Araneus [10], SIMS [11, 12]). But HTML has somehow to be reinvented for each wrapper.

On the other hand, a document hierarchy implied by tags can be used like DOM for [6] and [2], and at a higher-level for XML-Pointer [19]. However, navigation along this explicit structure is sometimes restricted to the hierarchy itself and cannot capture finer granularity information.

In WebL [8], the extraction is embedded as a piece of code (procedural description). Araneus [10] and SIMS [12] use a mix with a declarative grammar-based approach. In Web-OQL [3] and XML-QL [5] the extraction is a query with variable binding. WIDL [2], QEL[4] [6], XML-Pointer [19] and

[3]Such issues are the responsibility of the higher-level application.

[4]QEL: Qualified-path-expression Extractor Language.

XQL [15] offer very similar declarative approaches but cannot return complex constructs.

In W4F, we try to make the most of the HTML structure using the DOM object-model. This knowledge is a built-in feature of the system. It offers the power of regular expressions, some rich navigation capabilities with constraints and some constructs to access some finer grain information in order to capture as much structure (including nesting) as it can.

Mapping layer

Wrappers are in charge of providing a structured access to the extracted information. For Web-OQL [3], a Web document is an OQL instance from the beginning. In Tsimmis [7] the extracted information is converted into the OEM format (semi-structured data). [6] offers CORBA-like interfaces. YAT [16] offers an expressive rule-based framework (with a user-interface) to express mapping and transformations.

While some wrappers can directly build structured objects (complex objects, relations) out of the information, in W4F we prefer to separate extraction and manipulation and to use the NSL intermediate representation in order to favor re-use and tunability. The advantage of our *anonymous* NSL structure is that it can be re-used as is. Moreover, the extracted structure (e.g. the XML data of Figure 8) can be exported to languages like XML-QL [5].

Wrapper construction strategies

The manual generation of a wrapper often involves the writing of ad-hoc code ([7] and [10]). Web-OQL [3] takes advantage of a generic mapping between the HTML structure and the OQL object-model but it means writing complicated `select-from-where` queries.

Semi-automatic generation benefits from support tools to help design the wrapper. In Web-Methods [2], the entire structure understood by the system is presented to the user who has to pick what he wants. SIMS [12] offers a *demonstration-oriented* interface where the user shows the system what information to extract.

In [9], Kushmerick identifies some classes of wrappers on which he can apply machine-learning techniques to generate wrappers automatically. But the system has to be fed with training examples and might require human supervision.

In W4F, we rely on human expertise but offer support to make this creation accessible through some wizards (semi-automatic construction). The choice of the DOM object model gives us for free a real wysiwyg interface.

Parsing Web documents

Parsing HTML appears completely trivial at first sight, but most document found on the Web are non-compliant[5]. Parsers therefore need to be able to recover from ill-formed documents.

Since HTML is a subset of SGML, SGML parsers like James Clarke's SP can be used. The input is the HTML document and the corresponding DTD. An upgrade in the HTML version simply means providing the upgraded DTD. The cost of genericity is that such a parser may be quite big (1.2 Mb) and the recovery heuristics may not be targeted specifically to HTML.

Our parser is written using `Java CC`, compliant with HTML 3.2 specification with a footprint of less than 100kb. It allows the fine tuning that is really necessary to catch human misuses of the language. Our recovery heuristics are similar to some presented in [13].
From a performance point of view, parsing documents is more expensive than using regular expressions. Our parser's throughput is about 30kb/sec compared to 1Mb/sec for regular expression processing. The fetching of documents still remains the performance bottleneck for on-line application of the toolkit.

8 Conclusion and future work

In this paper we have presented the W4F toolkit to build wrappers for Web data sources. We have identified three independent layers (retrieval, extraction and mapping) that can be built independently so as to offer re-use and

[5]Since Web browsers are very permissive about it, people are not even aware that their documents are not HTML compliant.

rapid development.

For the extraction layer in particular, we offer a highly expressive extraction language that navigates the parse tree of the HTML document. It comes with some powerful operators to reach information at the tag level and deeper (multi-granularity), and permits to return some nested structures in order to truly capture the information expressed in the document in a robust way.

For each layer, we provide a declarative language to describe its semantics as well as some support tools (wizards) to help the user. The use of the Document Object Model gives us for free a wysiwyg interface used by the wizard.

We have been using successfully the toolkit to build a large range of wrappers for diverse Web resources like knowledge repositories (CIA World Factbook, MedLine, etc.), Web catalogues (Amazon.com, Comp-USA, etc.) and databases (Internet Movie Database, IBM Patent Server, etc.). Most of the time, we came up with a robust up-and-running wrapper within minutes thanks to our wizards. The maintenance has also been quite easy.

As part of some future work, we will focus on some improvement (robustness) of the rules returned by the extraction-wizard using some machine-learning technique. We also will investigate the notion of a Web service that offers a higher-level interface to Web data sources (object identity, caching, etc.). Finally, we will look at some optimization techniques for lazy-parsing[6] and rule evaluation in order to improve performance.

The W4F toolkit has been developed under JDK-1.1.5, using JavaCC to generate the the HTML and HEL. Regular expressions are evaluated using PAT, the regular expression package.

On-line examples of W4F applications can be found on the Penn Database Research Group web site.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *Journal on Digital Libraries*, 1997.

[6] The real challenge is to do lazy parsing while allowing recovery on ill-formed documents.

[2] Charles Allen. WIDL: Application Integration with XML. *World Wide Web Journal*, 2(4), November 1997.

[3] Gustavo Arocena and Alberto Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proc. ICDE'98*, Orlando, February 1998.

[4] Johnatan Crabtree, Scott Harker, and Val Tannen. An OQL interface to the K2 system. Technical report, University of Pennsylvania, Department of Computer and Information Science, 1998. To appear.

[5] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998.
URL: http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[6] Jean-Robert Gruser, Louiqa Raschid, María Exther Vidal, and Laura Bright. A Wrapper Generation toolkit to specifiy and construct Wrappers for Web Accessible Data Sources. Technical report, Institute for Advanced Computer Studies, University of Maryland, College Park, 1998.
URL: ftp://ftp.umiacs.umd.edu/pub/louiqa/BAA9709/PUB98/1CoopIS98.ps.

[7] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *Proceedings of the Workshop on Management of Semistructured Data. Tucson, Arizona*, May 1997.

[8] Thomas Kistlera and Hannes Marais. WebL: a programming language for the Web. In *WWW7*, Brisbane, Australia, 1998.
URL: http://www.research.digital.com/SRC/WebL/index.html.

[9] Nicholas Kushmerick. Wrapper induction: Efficiency and Expressiveness. Workshop on AI & Information Integration, AAAI-98, 1998.

[10] G. Mecca, P. Atzeni, P. Merialdo, A. Masci, and G. Sindoni. From Databases to Web-Bases: The ARANEUS Experience. Technical Report RT-DIA-34-1998, Universita Degli Studi Di Roma Tre, May 1998.

[11] Ion Muslea, Steven Minton, and Craig A. Knoblock. Wrapper Induction for Semistructured, Web-base Information Sources. Conference on Automated Learning and Discovery, June 1998.

[12] Naveen Ashish and Craig A. Knoblock. Semi-automatic Wrapper Generation for Internet Information Sources. In *Proc. Second IFCIS Conference on Cooperative Information Systems (CoopIS)*, Charleston, South Carolina, 1997.

[13] Dave Raggett. Clean up your Web pages with HP's HTML Tidy. In *WWW7*, Brisbane, Australia, 1998.
URL: http://www.w3.org/People/Raggett/tidy/.

[14] Arnaud Sahuguet and Fabien Azavant. W4F: the WysiWyg Web Wrapper Factory. Technical report, University of Pennsylvania, Department of Computer and Information Science, 1998. To appear.
URL: http://cheops.cis.upenn.edu/ sahuguet/WAPI/.

[15] David Schach, Joe Lapp, and Jonhatan Robie. XML Query Language (XQL), 1998. QL'98 - The Query Languages Workshop.

[16] Jérôme Siméon and Sophie Cluet. Using YAT to Build a Web Server. In *Proc. WebDB'98*, Valencia, 1998.
URL: ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-139.ps.gz.

[17] W3C. The Document Object Model, 1998.
URL: http://www.w3.org/DOM.

[18] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1996.

[19] World Wide Web Consortium. XML Pointer Language, 1998.
URL: http://www.w3.org/TR/1998/WD-xptr-19980303.

9 Appendix

9.1 HTML annotation for the form-wizard

```
function findForm(node) {
   if (node.tagName == "FORM") return node;
   else return findForm(node.parentElement);
}
function printForm(node) {
   var form = findForm(node);
   var str = "ACTION: " + form.action + "\nMETHOD: " + form.method + "\n";
   for(i=0; i<form.elements.length; i++) {
      if (form.elements[i].type != "submit") {
          str += "|\n+-- " + "NAME: \"" + form.elements[i].name
               + "\"  TYPE: \"" + form.elements[i].type
               + "\"  VALUE: \"" + form.elements[i].value + "\"\n";
        }
     }
   return str;
}
```

Figure 10: JavaScript extensions

The form annotation corresponds to the following transformation:
`<FORM >...<INPUT TYPE=submit ...>...</FORM>` becomes
`<FORM >...<INPUT TYPE=submit ...onMouseOver="alert(printForm(this))">...</FORM>`

9.2 HTML annotation for the extraction-wizard

The Wysiwyg interface takes a URL as an input, parses the corresponding HTML document and returns a new "annotated" HTML document.

The original document is fed into the W4F HTML parser to build a parse tree. Using the parse tree, the document is reconstructed in such a way that each chunk of text is now enclosed in a new tag generated by the parser with an ID value that corresponds to the path leading to it. A click on the chunk of text will display the ID value, i.e. the corresponding path.

The following annotation is performed: each PCDATA leaf corresponding to an outer tag is transformed. Basically, a new tag `<SPAN>` is inserted within

the outer tag. The benefit of it is that the chunk of text can now be given a specific behavior that can be used by the HTML browser.

Assuming that the path-expression corresponding to this tag in the HTML abstract tree is `"html.tag[n].txt"`, we have the following transformation: `<TAG> stuff </TAG>` becomes

```
<TAG><SPAN ID="html.tag[n].txt"> stuff </SPAN></TAG>
```

The new `<SPAN>` tag created now carries some information about the path that leads to this specific piece of information.

It is important to note that the path-expressions used for `ID` (i.e. returned by the interface) are *canonical* and only use the `"."` operator. This extraction rule might not be the most robust one but it is a good start anyway.