Looking at the Web through XML glasses

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

The Web so far has been incredibly successful at delivering information to human users. So successful actually, that there is now an urgent need to go beyond a browsing human *and make information accessible to applications, in order to offer automation, inter-operation and Web-awareness among services.*

To do so, information from Web sources needs to be accessible in a structured way. XML and its various extensions (data-models, query languages) are a step in this direction. Unfortunately, the Web is not yet a well organized repository of nicely structured XML documents but rather a conglomerate of volatile HTML pages, for which structure has to be extracted.

To address this problem, we present the World Wide Web Wrapper Factory (W4F), a Java toolkit for the generation of wrappers for Web sources. Our main contributions are: (1) an expressive language to specify the extraction of complex structures from HTML pages; (2) a declarative mapping to XML documents, with the automatic generation of the corresponding DTDs; (3) some visual supports to make the engineering of wrappers faster and easier.

As an illustration, we show how we can, via W4F intermediation, transparently query HTML sources from an XML query language.

1. Introduction

The Web has become a major conduit to information repositories of all kinds. Today, more than 80% of information published on the Web is generated by underlying databases (however access is granted through a Web gateway using forms as a query language and HTML as a display vehicle) and this proportion keeps increasing. But Web data sources also consist of stand-alone HTML pages hand-coded by individuals, that provide very useful information such as links, reviews, digests, etc.

An unfortunate consequence is that Web information sources exist independently of one another, like isolated information islands. For instance, when looking for a movie, one would like to combine movie details (genre, cast, etc.) from one site, with reviews from a second one, and show times from a third one. And today the only way to do it is by performing the clicking, fetching and reading by hand.

What one really would like is to go beyond a *browsing human*, in order to achieve automation, Web-awareness among services (services taking advantage of one another), interoperability (between Web sources and legacy databases or among Web sources themselves) and cooperation. All this requires software applications capable of making the content of HTML source available to them, via HTML wrappers.

This work is presented at an interesting time because of the on-going emergence of XML [17] as a more application-friendly standard for Web pages. Some people have already argued that there in no more need for HTML wrappers because data sources will soon serve XML documents. For data coming from information systems, the underlying databases can be easily modified to generate XML documents, but it might require some expensive changes. For HTML pages that have been hand-coded, the migration has to be handled using ad-hoc tools, if any. In both cases, HTML wrappers can smoothly perform data migration from these legacy information sources.

In fact, there already are countless HTML pages on the Web and the information that many of them contain will have to be displayed in XML in a relatively near future.

In this paper we present the World Wide Web Wrapper Factory (W4F), a toolkit to generate wrappers for HTML Web sources. Our wrappers are specified in a fully declarative way and handle transparently the retrieval of Web pages, the extraction of information from the HTML source and the mapping to a target format for further use. We argue that our toolkit can offer a simple and elegant solution to the problems of integration and migration, and we show how it can be used to *look at the Web through XML glasses*, by translating HTML pages into XML documents.

The rest of the paper is organized as follows. Section 2 briefly presents the W4F toolkit and the concrete problem we want to address. In section 3, we give an overview of the extraction language. Section 4 describes the internal data model used to store extracted data. Mappings and in particular mappings to XML are explained in Section 5. Section 6 presents the visual support offered by the toolkit. Section 7 suggests how we can use the toolkit to generate XML documents – either virtual or materialized – and directly query them with an XML query language. Finally we offer our concluding remarks with some future and related work.

2. W4F in a nutshell

W4F (World-Wide Web Wrapper Factory) is a toolkit to generate Web wrappers. Our wrappers consist of three independent layers. The **retrieval layer** is in charge of fetching the HTML content from a Web data source. The **extraction layer** extracts the information from the document. The **mapping layer**'s role is to specify how to export the data. These layers are working together as illustrated in Figure 1.

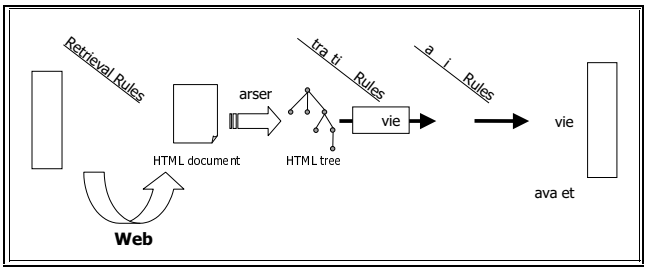


Figure 1. W4F information flow.

For a given Web source, some extraction rules and some structural mappings, the toolkit generates a Java class that can be used as a stand-alone program or directly integrated into a more complex application. A wrapper is specific to a class of Web pages. For the movie examples presented later on, it means that we need one wrapper to handle HTML pages for movies, one wrapper for actors, etc.

The toolkit per-se consists of an HTML parser that generates parse trees out of HTML pages (using various heuristics to handle ill-formed pages), a compiler to produce Java code for each layer and various visual wizards (see Section 6) to assist the user in writing the specifications.

Along this paper we will present how the toolkit can be used to integrate information from the Internet Movie Database (IMDb). IMDb is the biggest information repository about movies and is freely available. Its underlying information system is a big file system[1] that serves HTML pages.

[1] See http://www.imdb.com/interfaces#plain for more details.

Our concrete goal is to be able to ask queries about the *best movies*. IMDb proposes – among many other things – a page with the list of the best 250 movies (let us call it `Top250`) and a page for each movie (let us call it `Movie`). These pages are presented in Figure 2.

Figure 2. `Top250` and `Movie` pages.

As we can see, the `Top250` source offers limited information about movies: title and year only. To gather some extra information, we need to follow the link to the movie page where the detailed information is available. We therefore need to build two wrappers, one for each type of pages. In the rest of the paper, these wrappers will be implicitly referenced as `Top250` and `Movie`. From `Top250`, we will need to extract the title and the url of each movie in the list. From `Movie`, we will need to extract the title, the year, the list of directors, the list of genre and the cast.

In the next sections, we will present how to specify the information to be extracted from each HTML source, how to define a mapping to XML and how to enrich information from `Top250` with the details of each movie.

3. Extracting information

In this section, we present some features of HEL (HTML Extraction Language) used for the specification of the extraction layer. The full details can be found in [14]. Features presented here after are illustrated by the examples of Figure 3 and Figure 4.

HEL is a DOM-centric [18] language where an HTML document is represented as a labeled graph. Each Web

```
EXTRACTION_RULES for Top250
top250 = html.body.table[1].tr[0].td[2].div[0].table[0].tr[i:*].td[1] ( .a[0].getAttr(href) # .txt )
WHERE   html.body.table[1].tr[0].td[2].div[0].table[*].tr[i].getNumberOf(td) = 2;

EXTRACTION_RULES for Movie
movie = html
   ( .head.title.txt, match/(.*?) [(]/                                                    // title **
   # .head.title.txt, match/.*?[(]([0-9]+)[)]/                                           // year  **
   # .body.table[1].tr[0].td[2].table[0].tr[1].td[1].table[h:0].tr[1-].td[0].a[0].txt // directors
   # .body->td[i:0].a[*].txt                                                             // genres
   # .body->table[ii:0].tr[jj:*].td[0].txt, match/(\\S+)\\s(.*)/                         // cast
   )
WHERE html.body.table[1].tr[0].td[2].table[0].tr[1].td[1].table[h].txt =~ "Directed by"
AND   html.body->td[i].b[0].txt = "Genre"
AND   html.body->table[ii].tr[0].td[0].txt =~ "(Cast overview)|(credited cast:)"
AND   html.body->table[ii].tr[jj].getNumberOf(td) = 3;
```

Figure 3. Extraction rules for the IMDb expressed using HEL.

document is parsed and an abstract tree corresponding to its HTML hierarchy is built out of it. 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). Nodes can have children and these can be accessed using their label and their index. A leaf can be either a PCDATA or a *bachelor* tag[2].

Navigation along the abstract tree is performed using path-expressions ([5, 1]). A unique feature of HEL is that it comes with two ways to navigate.

The first navigation is along the **document hierarchy** using the "." operator. Path 'html.head.title' will lead to the node corresponding to the <TITLE> tag, inside the <HEAD> tag, from the root of the document. This type of navigation offers a unique (i.e. canonical) way to reach each information token.

The second way to navigate is along the **document flow**, using the "->" operator. Path 'html->pcdata[1]' will lead to the second chunk of text found in the depth-first traversal of the abstract tree starting from the root of the document. This operator is very useful to create navigation shortcuts. Moreover, it permits to traverse the entire tree.

Using both complementary navigation styles, most structures can be easily identified as extraction paths. To the best of our knowledge, HEL is the only language that captures both *structures* of a page.

Path expressions can also use **index ranges** to return a collection[3] 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.

For our extraction purposes, we are not really interested in nodes themselves but rather in the values 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.
In the same way, the underlying HTML source is extracted using ".src". Some other properties like attribute values (e.g. "HREF") or the number of children can also be retrieved from nodes.

Another key feature of the language is the ability to have path **index variables** that can be resolved with respect to some **conditions** when the path is evaluated on a given page. Index variables can return the first index value (like [i:0][4]) or an index range (like [i:*], for all of them) that satisfies the condition. Conditions are introduced using string variables and WHERE clauses separated by AND. Conditions cannot involve nodes themselves but only their properties. Various comparison operators are offered by the language, including regular expression matching.
Conditions can be marked with the cut operator "!"[5], meaning that the search for index values will be stopped the first time the condition is evaluated to false. This operator turns out to be extremely useful when used with "->" to limit the exploration of the tree.

Conditions are crucial in table contexts, where row and column positions are not known in advance for instance. Let us imagine we want to extract the name of the movie ranked 5^{th} and let us assume that the information is in a table with header "Best Movies" in which movies are presented one by row. For each movie, we have – among other information – a column "Title" and a column "Rank", as presented in Figure 4.

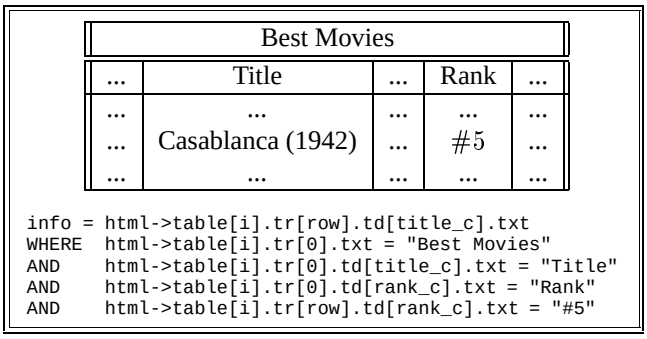
The table index i is resolved first, then column indices for "Title" and "Rank". Finally, we can resolve the row

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

[3] list, since we care about the order of nodes.

[4] This is the default behavior.

[5] In the spirit of the Prolog cut.



Best Movies				
...	Title	...	Rank	...
...	...	...	...	...
...	Casablanca (1942)	...	#5	...
...	...	...	...	...

```
info = html->table[i].tr[row].td[title_c].txt
WHERE  html->table[i].tr[0].txt = "Best Movies"
AND    html->table[i].tr[0].td[title_c].txt = "Title"
AND    html->table[i].tr[0].td[rank_c].txt = "Rank"
AND    html->table[i].tr[row].td[rank_c].txt = "#5"
```

Figure 4. Using index variables

that corresponds to rank `"#5"`. This is really an extreme case but it illustrates that using index variables, extraction does not require to know the exact structure ahead of time. In most real examples (see Figure 3), we already know some partial information about the table structure.

So far we have been using only the HTML hierarchy to extract information. However, in many cases, the tag granularity is too rough and we need something thinner to capture more precise information. For instance, in the table example of Figure 4, we might want to extract the title itself and trim the year.
To capture this level of details, our language comes with standard **regular expressions** à la Perl [16] that can be accessed through the two operators `match` and `split`. The `match` operator takes a string and a pattern, and returns the result of the matchings (there can be more than one). Depending on the nature of the pattern[6] the result can be a string or a list of strings.
The `split` operator takes a string and a separator, and returns a list of substrings. These operators can also be used in cascade.
In the example of Figure 3, `match` is used to extract separately the title and the year from the movie page title `'head.title.txt'`. `split` would be used when for instance the information is returned as a string with a delimiter like `","`, `"-"` or `"and"`.

As pointed out previously, extraction should not be limited to isolated pieces of information but should be able to capture **complex structures**.
The HEL language therefore provides the fork operator `"#"` to build complex structures based on extraction rules. The meaning of the operator is somehow to follow multiple subpaths at the same time. Forks can be applied in cascade. This is particularly useful when information spread across the page need to be put together like in the movie examples of Figure 3.

[6] The number of parenthesized sub-pattern binders indicates the number of items returned by the match. `(?: )` is not a binder!

4. Storing information as NSLs

A key motivation of W4F is to be able to capture complex structures expressed inside HTML pages. The extraction language presented in the previous section offers rich constructs, but we also need a flexible and expressive way to represent the extracted information.

Within W4F, information is stored in *Nested String List*s (NSL), the data-type defined by `null | string | list of NSL`. It is important to note that items within a list can have different structures. The data-type has been chosen on purpose to be simple, anonymous and capable of expressing any level of nesting.

For a given extraction rule, the structure of the corresponding NSL is fully determined by the rule itself (the `WHERE` clause has no influence). Strings are created by leaves. Lists are created from index ranges, forks and regular expression operators `split` and `match` (only when the number of matches is greater than one).

By looking at the extraction rules of Figure 3, we can infer that for a `movie` the corresponding NSL will be a list of 5 items (4 top level forks). Items 0 and 1 will be strings (1 match with 1 binding). Item 3 will be a list of strings (index range `tr[1-]`). Item 4 will be a list of strings (index range `a[*]`). Item 5 will be a list of pairs (index range `tr[jj:*]` and 1 match with 2 bindings).
For `top250`, the NSL will be a list (`tr[i:*]`) of pairs (1 fork) of strings.

NSLs are very low-level structures that can be manipulated via an API (list iterators and coercion operators). For the example of the best 250 movies, we get the NSL from a call to the wrapper (`nsl = getTop250()`) and then extract the various components as presented in Figure 5.

```
NestedStringList nsl = getTop250();
String url, title; NSL_List l;
for(int i=0; i<nsl.size(); i++)
{
 l     = (NSL_List)nsl.elementAt(i);
 url   = l.elementAt(0).toString();
 title = l.elementAt(1).toString();
 }
```

Figure 5. Using the NSL API.

5. Mapping information

As presented above, the only way to transform NSLs is to use the low-level API. Fortunately, W4F comes with automatic mappings for various target types that handle the low-level coding for the user: the user can simply invoke

the mapping, the NSL coding being done transparently under the hood.

5.1. Java mappings

First, W4F offers an automatic mapping to Java base types and their array extensions. Strings, int, float can be automatically coerced into the corresponding types. Nesting is handled, but only for homogeneous structures (list items must have the same structure).
For more irregular structures, the user has to define a Java mapping by providing some Java classes with valid constructors that can consume (via the API) the NSL to produce Java class instances. This is a very powerful construct that allows the user to do almost everything from the information carried by the NSL. We present below a way to map the NSL from the `Top250` wrapper into a `MovieRef` object. W4F will automatically convert the NSL extract from the page into an array of `MovieRef`.

```
public class MovieRef
{
  String url, title;
  public MovieRef( NestedStringList nsl )
  {
    NSL_List pair = (NSL_List) nsl;
    url   = pair.elementAt(0).toString();
    title = pair.elementAt(1).toString();
  }
}
```

Figure 6. Defining a mapping to a Java class.

Using the same strategy, the user could define some Java classes that offer some XML output for instance. But in this case the mapping would be defined in terms of code.

5.2. XML mappings with XML templates

An XML mapping expresses how to create XML elements out of NSLs. It is important to keep in mind that the *shape* of XML elements we can generate is constrained by the structure of the NSL itself.

An XML mapping is described via declarative rules called *templates*. Templates are nested structures composed of *leaves*, *lists* and *records* and are defined using the language defined below:

Template	:=	Leaf | Record | List
Leaf	:=	. Tag | . Tag ^ | . Tag ! Tag
List	:=	. Tag Flatten Template
Record	:=	. Tag **(** TemplList **)**
Flatten	:=	* | * Flatten
TemplList	:=	Template | Template # TemplList
Tag	:=	string

We detail next each type of template. The XML elements and the DTD specification that correspond to each template are presented in figures 7, 8 and 9.

A **leaf template** consumes an NSL that is a string. Various target XML elements can be desirable. The string can be represented as `PCDATA`, as an attribute of a parent element or as attribute of a bachelor element. We show below how these cases can be specified in the language[7] and the output XML element for an input NSL string (say "Se7en"):

```
.Movie
<!ELEMENT Movie #PCDATA>
  <Movie>Se7en</Movie>

.Movie( .Title^ # _ )
<!ELEMENT Movie (_)>
<!ATTLIST Movie Title CDATA #IMPLIED>
  <Movie Title="Se7en" __ > _ </Movie>

.Movie!Title
<!ELEMENT Movie EMPTY>
<!ATTLIST Movie Title CDATA #IMPLIED>
  <Movie Title="Se7en"/>
```

Figure 7. Leaf templates.

A **list template** like .Movies*.templ consumes a list of NSL items. It first opens a new element `<Movies>`. Then it applies the same template templ to each list item, using concatenation. Finally the element is closed with `</Movies>`. In the list template, the number of `'*'` indicates if any flattening has to be performed on the NSL list, before applying the template.

```
.Movies*.templ
<!ELEMENT Movies (templ)*>
  <Movies>
    <templ> ...  </templ>
    ...
    <templ> ...  </templ>
  </Movies>
```

Figure 8. List template.

A **record template** like .Movie(t_1 # __ # t_n) consumes a list of n NSL items. It first creates a new element `<Movie>` and applies each inner template to its corresponding list item, using concatenation. Finally the element is closed with `</Movie>`.
For a record, a different template is applied to each NSL item; for a list, it is the same template.

From an XML mapping, W4F will generate some Java code that represents a template. The template can later on be used to consume the NSL and produce XML documents.

[7] The sequence __ in the lines below simply means "anything".

.Movie(T1 # _ # Tn)

```
<!ELEMENT Movie (T1, ..., Tn)>
  <Movie>
    <T1> ...  </T1>
    ...
    <Tn> ...  </Tn>
  </Movie>
```

Figure 9. Record template.

The construction of the DTD is straightforward from the specification itself.

Some important remarks about the mapping are worth mentioning.

The mapping is directed by the extraction
A mapping is a way to consume the NSL and a NSL piece can only be consumed once. If the user wants to have an `Actor` element with two sub-elements `FirstName` and `LastName` and an attribute `Name`, he must make sure that the NSL carries these three items. For a given purpose, it might be necessary to change the extraction rule, to come up with the desired XML element.
There is no 1-1 mapping between a template and an NSL. First, the user has some freedom when defining leaves for instance. Second, record and list templates both consume NSL lists that may have been created by different means: `html__(.tag[0].txt # .tag[1].txt)` and `html__.tag[*].txt` can both be consumed by the same record template provided that the second NSL consists of two elements.
Another issue is that templates do not capture the full expressivity of XML DTDs. For instance, it is not possible to define IDREFs.

A template corresponds to more than one DTD
As mentioned above, the creation of a DTD from a template is straightforward. However, for list and record templates the generated DTD can be set to be more or less permissive. As presented in figures 7, 8 and 9, the DTD is permissive for lists (using a * that means 0 or more elements) and strict for records (sub-elements have to appear). The latter policy could be weakened by replacing `(T1,...,Tn)` with `(T1?,...,Tn?)`. These choices depend on the use that will be made of the DTD (optimization, validation) and on the quality (in terms of regularity of the structures) of the web source from where the information is extracted. For the IMDb example, the record policy turns out to be too restrictive if we want to guarantee the validity of the XML documents generated.

Templates do not prevent *pathological cases*
The `"^"` construct should not be used in a list environment. Movies*.Titleˆ will try to add attribute `Title` to element `Movies` for each element of the NSL list being consumed. The semantics is that the attribute value will be overwritten by every next element.
XML requires that a tag must be defined only once. In the movie template, changing `Title` and `Genre` to `Name` will be fine since the new element `Name` will be the same in both contexts. However, changing `Director` and `Actor` to `Person` will create a conflict: `Person` as a sub-element of `Directed_By` only contains `PCDATA` while `Person` as a sub-element of `Actors` contains `FirstName` and `LastName`.

```
movie_t =
  .Movie ( .Title
         # .Year
         # .Directed_By*.Director
         # .Genres*.Genre
         # .Cast*.Actor ( .FirstName
                        # .LastName )  );
```

Figure 10. An XML template for a movie.

As an illustration, we present in Figure 10 a possible XML mapping for the IMDb Web source. The output of this mapping will appear in Figure 14 and its DTD in Figure 15.

6. Support via visual tools

In order to make the specification of our wrappers fast and easy, W4F provides some visual tools (or wizards) that assist the user during the various stages of the wrapper construction.

The critical part of the design of the wrapper is the definition of extraction rules since it requires a good knowledge of the underlying HTML.

Figure 11. The extraction wizard on `Top250`.

The role of the extraction wizard (see Figure 11) is to help the user write such rules. For a given HTML document, the wizard feeds it into the HTML parser and returns the document to the user with some invisible annotations (the document appears exactly as the original).

On Figure 11, when the user points to "Casablanca", the corresponding text element gets high-lighted (the user can identify information boundaries enforced by the HTML tagging) and the canonical[8] extraction rule pops-up.

The magic behind it takes advantage of our DOM-centric approach: the page is fed into the parser and each text chunk (i.e. PCDATA) gets annotated with its corresponding canonical path in the document tree.

As an example, the annotation of a tree node with label TAG (which is the n^{th} child of root node HTML) is given below:

```
<TAG> stuff </TAG>
```

becomes

```
<TAG>
<SPAN ID="html.tag[n].pcdata[0].txt">
stuff
</SPAN>
</TAG>
```

This straight-forward annotation strategy carries some restrictions. First, the path used is the canonical path: it does not use all the powerful constructs of the HEL language like "->", index ranges, conditions or regular expressions. Secondly, the annotation is done for each element. In the example of Figure 11, it would be convenient to point to all the items from the list, not just one.

But even if the wizard is not capable of providing the best extraction rule, it is always a good start. Compare what is returned by the wizard and what we actually use in our wrapper for Top250 (Figures 11 and 3).

Another useful interface permits to test and refine the wrapper interactively before deployment. Figure 12 shows the wizard which visualizes the 3-layer architecture of the wrapper.

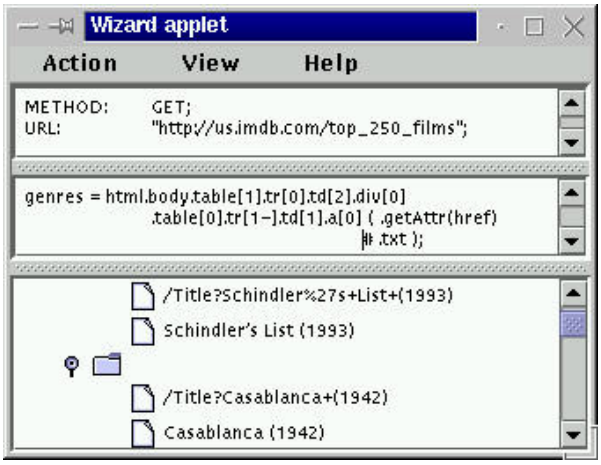


Figure 12. A visual view of the wrapper.

[8] By canonical we mean that it uses only hierarchy based navigation.

In the top layer, the user inputs the location of the Web source and the retrieval method (a GET by default). The middle layer displays the extraction rule – expressed in the HEL language – to be applied on the retrieved HTML page. In this example, the rule will extract the url and the full title (title and year) of each movie of the list. The bottom layer represents the structure of the information extracted. In the case of the figure, the NSL data-structure returned is a list of pairs of strings.

As of this writing, a mapping wizard is under construction.

7. Querying HTML sources

Using the wrappers presented above, we can now extract information from HTML pages and map it into XML documents.

7.1. Materialized vs. virtual documents

From an engineering perspective, two cases can occur: (1) the XML document is generated out of one unique HTML page; (2) the XML document is generated out of many HTML pages.

In the first case, the translation can be done on-the-fly and through an XML gateway. The role of the gateway is simply to offer a Web access (via a parameterized url u) to the page: from u it extracts the location of the source page and performs the extraction and the mapping. The access appears completely transparent for the caller.

In the second case, the translation requires HTML pages to be fetched and processed before being mapped together to the target XML document. As an illustration, we present in Figure 13 the piece of Java code that creates the XML document with the best movies and their detailed information.

```
XmlDoc doc = new XmlDoc();
doc.appendDTD_multiple("List", movie_t);
MovieRef[] refs = get_top250();
for(int i=0; i<refs.length; i++)
 doc.appendElement(getMovie(ref[i].url));
doc.print(new PrintWriter(System.out));
```

Figure 13. Materializing Top250.

We first prepare our new XML document that will be built out of multiple sources. Sources will be mapped according to the movie_t template and glued together under an element with name "List". Then we extract the information from the Top250 page and map it into the MovieRef Java structure defined in Section 6. We then iterate over each MovieRef and fetch the page correspond-

ing to its URL. The information is extracted, mapped into an XML element according to template `movie_t` and pushed to the XML document. The final XML document is presented in Figure 14 with its DTD in Figure 15.

Figure 14. The translated XML document ...

```
<!DOCTYPE W4F_DOC [
  <!ELEMENT W4F_DOC (List)>
  <!ELEMENT List (Movie)*>
  <!ELEMENT Movie (Title,Year,
         Directed_By,Genres,Cast)>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Year (#PCDATA)>
  <!ELEMENT Directed_By (Director)*>
  <!ELEMENT Director (#PCDATA)>
  <!ELEMENT Genres (Genre)*>
  <!ELEMENT Genre (#PCDATA)>
  <!ELEMENT Cast (Actor)*>
  <!ELEMENT Actor (FirstName,LastName)>
  <!ELEMENT FirstName (#PCDATA)>
  <!ELEMENT LastName (#PCDATA)>
]>
<W4F_DOC>
```

Figure 15. ... and its DTD.

7.2. Querying the XML documents

Once we have our XML document available (virtual or materialized), we can query them using our favorite XML query language. The following examples presented in figures 16 and 17 are expressed using XML-QL. See [8] for more details about the query language itself.

`Query1` looks for people who played in all the three Star-Wars movies. The query will use virtual XML documents that are generated on-the-fly from the movie page of each of the Star-Wars movies. `"...Star+Wars+(1977)"` represents the parameterized url that is used to access the virtual document.

```
Query1::
CONSTRUCT <Actor> <Name> $name </> </>
WHERE
  <*.Cast.Actor.LastName> $name </>
IN "...Star+Wars+(1977)",
  <*.Cast.Actor.LastName> $name1 </>
IN "...Empire+Strikes+Back,+The+(1980)",
  <*.Cast.Actor.LastName> $name2 </>
IN "...Return+of+the+Jedi+(1983)"
text($name) = text($name1),
text($name) = text($name2)
```

Figure 16. Query1.

`Query2`[9] looks for movies and pairs of people who worked together in more than one movie. The query will compute the cross-product for movies and then for each pair of movies the cross product of actors. Conditions make sure the same actor is not counted twice and that a pair of actors is only counted once (`$x < $y`). The query will use the materialized view of `Top250` presented before.

8. Conclusion and future work

In this paper we have presented the W4F toolkit and showed how it can be used to convert HTML pages into XML documents, by specifying some extraction rules (what information to extract) and a mapping to XML (what XML elements to create). From this perspective, our main contributions are: (1) a fully declarative specification of all the components of a wrapper; (2) a very expressive extraction language based on the Document Object Model, with two types of navigation, variables, conditions, regular expressions and some constructs to build complex structures; (3) a simple specification to map the extracted information into XML elements; (4) a robust framework to engineer wrappers for Web sources, that offers the generation of ready-to-use Java classes and some visual tools to assist the user.

Compared to other approaches [11, 12], we do not use a grammar-based approach for extraction but rely on the

[9] The authors are grateful to Alin Deutsch for working out the query and making sure it actually runs under the XML-QL implementation.

```
Query2::
CONSTRUCT <Joint_Work ID=f($x,$y)>
  <TITLE>$title1</> <TITLE>$title2</>
  <Actor>$x     </> <Actor>$y     </>  </>
WHERE <*.Movie>
         <*.Title>$title1</>
         <*.Actor.LastName>$a11</>
         <*.Actor.LastName>$a12</>
      </> IN "Top250.xml",
      <*.Movie>
         <*.Title>$title2</>
         <*.Actor.LastName>$a21</>
         <*.Actor.LastName>$a22</>
      </> IN "Top250.xml",
  text($title1) != text($title2),
  text($a11) = $x, text($a12) = $y,
  $x < $y,
  text($a21)    != text($a22),
  text($a21) = $x, text($a22) = $y
```

Figure 17. Query2.

DOM object-model, which gives us for free some wysiwyg visual tools like [2].

With rich features like hierarchical and flow-based navigations, conditions and nested constructs, our extraction language is more expressive and robust than [9, 3]. Unlike [4], we do not try to query the Web (for instance it is not possible to follow links at the level of the HEL language) but simply extract structure from Web information sources: querying is the concern of the application as illustrated by the XML-QL examples.

Our tackling of XML is different from the one of XML-QL [8] based on patterns and explicit constructs because we derive it from our extraction process that handles HTML pages with no explicit structure. As a consequence we cannot for instance express object sharing (using `IDREF`s). From this perspective, our approach is more related to XQL [15].

Similarly in W4F, we do not address problems that are specific to mediators but we believe that our wrappers can be easily included into existing integration systems like TSIMMIS [10], Kleisli [7], Garlic [13], etc. W4F is now being used as part of the K2 [6] integration system.

Our future work will involve three distinct aspects of the system. The first one focuses on providing better tools to assist the user when writing extraction rules. Machine-learning techniques should be useful to define robust shortcuts for complicated extraction paths. The second aspect concerns the mapping of NSL structures: extending the XML template language to handle `IDREF`s and providing other mappings for other data-models. For this last point, the solution might be to rely on XML and mappings from XML to other data-models. The third aspect is to enrich the language with some extraction patterns and some user defined functions. The possibility of following hyperlinks at the level of the extraction language has to be investigated: it permits to put two wrappers in the same extraction rule, but it forces to look at a page as a graph and not as a tree. We also have to make sure that the internal parser keeps up with new evolutions of the HTML language and recovers from new misuses by human authors.

The W4F has been successfully used to build a large variety of Web wrappers for information sources like the CIA World Factbook[10], the IBM Patent Server [11], Hoover's company profiles [12]. The toolkit can be downloaded from the Penn Database Research Group web site [13]. On-line examples of W4F applications (including the wrapper presented here) can be found at the same location.

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.

[2] Brad Adelberg. NoDoSE – A Tool for Semi-Automatically Extracting Semi-Structured Data from Text. In *Proc. of the SIGMOD Conference*, Seattle, June 1998.

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

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

[5] Vassilis Christophides. *Documents structurés et bases de données objet*. PhD dissertation, Conservatoire National des Arts et Metiers, October 1996.

[6] 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, 1999. To appear.

[7] Susan Davidson, Christian Overton, Val Tannen, and Limsoon Wong. Biokleisli: A digital library for biomedical researchers. *Journal of Digital Libraries*, 1(1):36–53, November 1996.

[10] http://www.odci.gov/cia/publications/factbook
[11] http://www.patents.ibm.com
[12] http://www.hoovers.com
[13] http://db.cis.upenn.edu/W4F

[8] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998. http://db.cis.upenn.edu/XML-QL.

[9] Jean-Robert Gruser, Louiqa Raschid, M. E. Vidal, and L. Bright. Wrapper Generation for Web Accessible Data Sources. In *COOPIS*, 1998.

[10] 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.

[11] Gerald Huck, Peter Fankhauser, Karl Aberer, and Erich J. Neuhold. JEDI: Extracting and Synthesizing Information from the Web. In *COOPIS*, New-York, 1998.

[12] 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.

[13] Mary Tork Roth and Peter Schwartz. A Wrapper Architecture for Legacy Data Sources. Technical Report RJ10077, IBM Almaden Research Center, 1997.

[14] Arnaud Sahuguet and Fabien Azavant. W4F, 1998. http://db.cis.upenn.edu/W4F.

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

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

[17] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[18] World Wide Web Consortium (W3C). The Document Object Model, 1998. http://www.w3.org/DOM.