University of Pennsylvania
ScholarlyCommons

Database Research Group (CIS) | Department of Computer & Information Science

3-1-2001

Building Intelligent Web Applications Using Lightweight Wrappers

Arnaud Sahuguet
University of Pennsylvania

Fabien Azavant
École Nationale Supérieure des Télécommunications

Postprint version. Published in *Data and Knowledge Engineering*, Volume 36, Issue 3, 2001, pages 283-316.
Publisher URL: http://dx.doi.org/10.1016/S0169-023X(00)00051-3

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/db_research/18
For more information, please contact repository@pobox.upenn.edu.

Building Intelligent Web Applications Using Lightweight Wrappers

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. Unfortunately, the Web is not yet a well organized repository of nicely structured documents but rather a conglomerate of volatile HTML pages. To address this problem, we present the World Wide Web Wrapper Factory (W4F), a toolkit for the generation of wrappers for Web sources, that offers: (1) an expressive language to specify the extraction of complex structures from HTML pages; (2) a declarative mapping to various data formats like XML; (3) some visual tools to make the engineering of wrappers faster and easier.

Keywords
web, XML, information extraction, wrappers

Comments
Postprint version. Published in *Data and Knowledge Engineering*, Volume 36, Issue 3, 2001, pages 283-316.
Publisher URL: http://dx.doi.org/10.1016/S0169-023X(00)00051-3

This journal article is available at ScholarlyCommons: http://repository.upenn.edu/db_research/18

Building Intelligent Web Applications Using Lightweight Wrappers

Arnaud Sahuguet [a], Fabien Azavant [b]

[a] *Department of Computer and Information Science, University of Pennsylvania Moore School Building, 200 South 33rd Street Philadelphia, PA 19104-6389, USA*

[b] *Ecole Nationale Supérieure des Télécommunications 46, rue Barrault, Paris 75634 Cedex 13, France*

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. Unfortunately, the Web is not yet a well organized repository of nicely structured documents but rather a conglomerate of volatile HTML pages.

To address this problem, we present the World Wide Web Wrapper Factory (W4F), a toolkit for the generation of wrappers for Web sources, that offers: (1) an expressive language to specify the extraction of complex structures from HTML pages; (2) a declarative mapping to various data formats like XML; (3) some visual tools to make the engineering of wrappers faster and easier.

Keywords: web; XML; information extraction; wrappers

1 Introduction

The Web has become a major conduit to information repositories of all kinds. Because it is based on open standards, has low entry costs for publishers and offers free navigation tools for end-users, it has become the de-facto standard for publishing information. It allows at the same time individuals, companies, independent and governmental organizations to publish information – for research, fun, profit – at a very low cost. Individuals create Web sites dedicated to their hobbies. Companies put on-line annual reports, catalogues, marketing brochures, product specifications. Government agencies publish new regulations, tax forms, etc. Independent organizations make available latest research results (e.g. the Human Genome Project). As of today, for some specific domains, the "reference" information can only be found on the Web and this is even truer for real-time data such as stock-market (e.g. The New-York Stock Exchange or NASDAQ), weather forecasts, etc.

Information on the Web consists of multimedia components (pictures, movies, sounds, applets, etc.) glued together in pages (documents). These documents are interconnected by hyperlinks and vary from pages generated on-the-fly by computer programs or database systems, to stand-alone pages hand-crafted by individuals. Both categories offer valuable information such as links, reviews, digests, etc.

All Web information sources have two things in common: (1) text content is delivered using HTML; and (2) access to content is made available through browsing (hopping from hyperlink to hyperlink) or form-based querying.

First, HTML has been mainly designed to tag information for display purposes and is not suitable to represent structure: HTML tags are more concerned with font size, color, position, etc. The structure of a document (if any) is defined *implicitly* by these tags. As consequence, for data coming from underlying databases and published on the Web, its structure is lost in the transformation from a database record into an HTML document, and to recover the structure, one has somehow to *reverse engineer* it from the HTML.

Second, access to content is twofold: browsing and querying. *Browsing* means that a document contains some links to some other documents: by navigating pointers, it is possible to reach some specific information (like following a path in a file system). *Querying* means that it is possible to go directly to a document: the navigation has been shortcut. It is important to make the distinction between Web querying from the traditional database querying. Web querying is versatile, in the sense that similar queries are not guaranteed to offer similar results. For instance, when looking up a book from an on-line bookstore, depending on the input title, one can get the description of the book, a list of candidate matches or an empty result. This is radically different from a database query where the *type* of the result is always known in advance.

1.1 Challenges

• **Automation.** Human users are now overloaded with Web information. Services like the AltaVista search-engine are terribly useful, but how many users have enough patience to go through the tens of Web pointers returned for a given query. For each of them, the human user has to click on the link, wait for the page to be downloaded to the browser and read the content: the entire process is done by hand. It is now crucial to have some tools to automate Web information processing on behalf of the human user. By automation, we do not necessarily mean the need for a heavy machinery: many automation issues do not concern huge amount of data but an amount of data it is too tedious to manipulate by hand like filtering hundreds of results from an AltaVista query, comparing dozens of products from an on-line catalogue, etc.

• **User-friendly vs application-friendly.** The Web is now being used as a medium of communication for humans but also for computer applications. It

has to evolve from a user-friendly only medium to a both user- and application-friendly one (the push towards XML is an attempt to solve the problem). The development of E-Commerce (both Business-to-Consumer and Business-to-Business) and Electronic Document Interchange will see computer programs exchange information using the Web.

• **Value added networks.** The future of the Web is already focused on value added networks (VAN) that aggregate information from various sources and offer better access, better analysis, etc. Search engines and portals are first attempts in this direction. However, accessing the data in order to enhance it is a challenge.

1.2 Web Applications

Web applications aim to add value to Web data that is (largely and freely) already published. They are the means for automating information processing on behalf of the user and creating the valued added networks we have discussed above. The process of adding value consists of pulling the data together from various sites, then filtering, comparing, and analyzing it, and finally publishing the results of the analysis as new Web data. The resulting synthetic information is likely to become in turn raw data for other Web applications!

These applications have to cope in particular with the following constraints, inherent to Web environments: uniform access, scalability, evolution, composability and autonomy.

• **Autonomy.** Applications cannot make strong assumptions about Web sources. The latter are unlikely to be modified just for the sake of one application. Yahoo! is not willing to change its quote services to make it easier for computer programs to extract quote values. Web content has to be accessed as it is presented to the human user.

• **Composability.** Web applications will consist of small components that can be assembled together. A good analogy is Unix shell scripts – that are very simple programs that can be combined to perform smart processing – or GUI components. Web applications should be lightweight and portable in order to be run in diverse environments from desktop to nomad computing devices.

• **Evolution.** Evolution is a key in this Web environment in perpetual motion. Web applications need to rely on abstractions and interfaces that can be modified quickly and independently, in order to support the versatility of the Web.

• **Scalability.** Processing can be split into simpler tasks that can be resolved in a distributed way. Web applications need to be built around Web APIs that offer a transparent access to Web data.

• **Uniform Access.** Web applications have to use Web standards in order

to access and serve information.

1.3 Overview of this paper

In this paper, we present an approach for the design of Web applications that relies on the *World Wide Web Wrapper Factory* (W4F), a toolkit for the rapid design, generation and integration of Web wrappers.
The rest of the paper is organized as follows. In Section 2, we give an overview of the W4F approach. Sections 3, 4, 5 and 6 present informally details of the toolkit, including the HEL language used to extract information from HTML documents, our internal data-model, some mappings to other data-format and the visual support we offer with the toolkit.
In Section 7, we give a concrete example of a Web application for information integration, that takes advantage of the W4F approach. Section 8 describes other examples of applications and experiences with W4F. Some related work is presented in Section 9 before we offer some concluding remarks. The formal description (denotational semantics) of the core of the HEL extraction language is presented in the appendices.

Along this article, examples will be motivated by the case-study of Section 7 that involve movie and TV program resources. Movie information will be extracted from the Internet Movie Database [1] (IMDb). TV program information will be extracted from the Yahoo! TV Coverage [2].

2 The W4F Approach

In this section we present the approach we use to build Web applications. It is based on a middleware [29] architecture with Web wrappers as illustrated in Figure 1. Wrappers – also often called adapters – are computer programs that offer high-level view and access to some data. Using them, the data can be handled transparently in a uniform and structured way. We qualify our wrappers *lightweight* because they are meant to execute simple tasks, require little resources and are defined in a concise way. The role of a wrapper is to offer to mediators an access to information that is independent of the structure of the source (HTML formatting in the case of Web wrappers). Mediators can later on export an enriched view of the data to clients. For instance a mediator will offer a unified view over multiple Web sources.

Key issues when dealing with Web sources are versatility and scalability. It is important to have tools that make the generation and maintenance of such wrappers easy. The *World Wide Web Wrapper Factory* (W4F) is a develop-

[1] IMDb is the biggest information repository about movies and is freely available at http://www.imdb.com.
[2] http://tv.yahoo.com

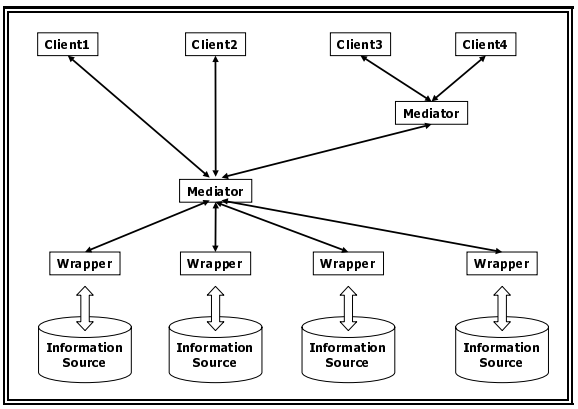

Fig. 1. Middleware architecture with wrappers and mediators.

ment environment that permits application developers to author a wrapper using a declarative specification language, compile it as a Java component and deploy it as part of a bigger application. The toolkit also offers some visual wizards to assist him during the design, testing and deployment of the wrapper.

Our Web wrappers are in charge of four independent tasks: retrieving a Web document, cleaning it, extracting some information from it and mapping this information into a pre-defined data-structure for further use. The details of these interactions are presented in Figure 2.

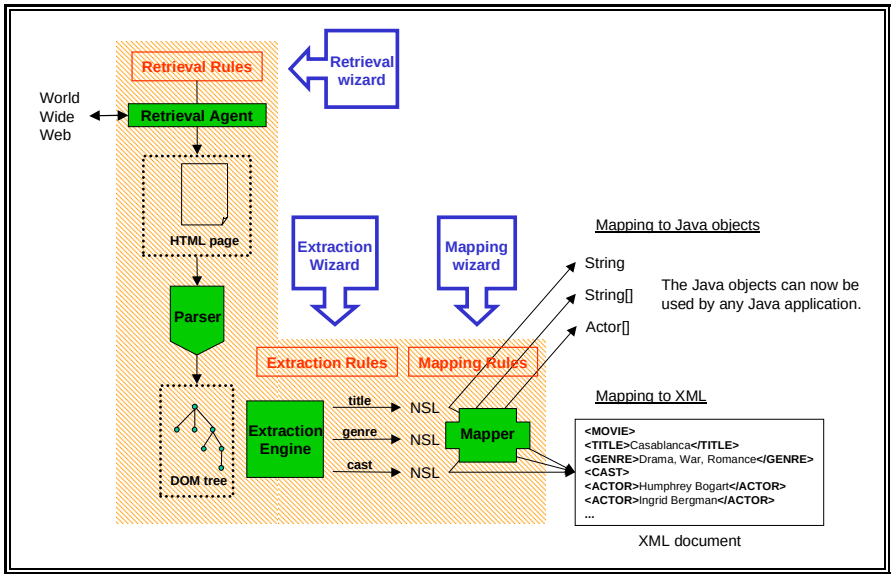

Fig. 2. W4F architecture

• **Retrieving a Web document.** This task is simply to mimic the action of a human fetching the page from his Web browser. Any page that can be accessed by a human is accessible to the wrapper. The retrieving is performed by our *RetrievalAgent* using the HTTP protocol.

• **Cleaning.** Unlike the XML specification [30] that enforces some constraints on the syntax of XML documents (well-formedness, validity), HTML has been

"hijacked" by users: a "good" HTML document is simply a document that "looks good" when viewed in a Web browser. Unfortunately, HTML documents are often not well-formed, in the sense that tags are not always properly nested (missing closing tags or overlapping tags). The cleaning stage transforms the HTML document into a well-formed document, that can be mapped in a DOM [31] tree.

• **Extracting information.** Once the HTML document has been retrieved and cleaned, it is parsed and an abstract tree representation is built out of it. Some extraction rules are then applied to extract some pieces of information from the tree.
Extraction rules are expressed using our high-level extraction language HEL (HTML Extraction Language). An extraction rule will express a navigation along the tree and will specify which pieces of information to collect and how to put them together. It is important to understand that an extraction rule simply expresses some interest for a piece of information in the document but does not mention anything about how this piece of information has to be used. The extracted information is stored in our internal data-structure (*nested string list* or *NSL*) before being used.

• **Mapping information.** The information extracted and stored in our internal representation is still not really usable and need to be mapped into an exportable structure suitable for the application.
The toolkit offers various ways to define mappings from our internal representation into user-defined data-structures, via either a declarative or a programmatic mapping specification.

Using W4F, we can now describe a wrapper in a fully a declarative way. The specification consists of 3 sections that correspond to the 3 layers mentioned above. An example of such a specification is presented in Figure 3.

```
SCHEMA
{
String title;
int year;
String[] genres;
String[][] cast;
}
EXTRACTION_RULES
{
title  = html.body->h1.txt, match/(.*?) [(]/;
year   = html.body->h1.txt, match/.*?[(]([0-9]+)[)]/;
genres = html.body->td[i:0].a[*].txt
  WHERE  html.body->td[i].b[0].txt = "Genre";
cast   = html.body->table[i:0].tr[j:*].td[0].txt, match/(\S+)\s(.*)/
  WHERE  html.body->table[i].tr[0].td[0].txt =~ "Cast"
    AND  html.body->table[i].tr[j].getNumberOf(td) = 3;
}
RETRIEVAL_RULES
{
   get(String url) { GET "$url$"; }
}
```

Fig. 3. The full wrapper specification for Internet Movie Database (IMDb).

The `RETRIEVAL_RULES` section defines methods to access the Web source. In the example, a valid movie url[3] needs to be provided. This is one mandatory input of the wrapper.
The `EXTRACTION_RULES` section defines what information to extract from the Web source. An extraction rule consists of (1) a name that is used to refer to this specific data and (2) an HEL extraction path.
The `SCHEMA` section defines the mapping, i.e. how extracted elements will be available from the wrapper. In the case of the figure, the wrapper will export a title as a Java `String`, a year as an `int`, a list of genres as a `String[]` and a cast as `String[][]`.

A wrapper is specific to a class of Web pages. For the examples presented in upcoming sections, we will need one wrapper to handle HTML pages for the TV program, one wrapper to handle HTML pages for movies, etc. Now, for a given Web source, the specification is compiled into a Java component that can be used as is or integrated in a larger application. 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. The various components of the toolkit are described in more details in the following sections.

3 The HTML Extraction Language (HEL)

In this section, we describe informally some features of HEL (HTML Extraction Language) used for the specification of the extraction layer. The full syntax of the language is available in [5]. A formal description of the core language can be found in Appendix A. Features presented here after are motivated by the examples of figures 3, 6 and 7.
HEL is a DOM-centric [31] language where a document is represented as a labeled graph. In this article, we will use HEL to navigate HTML documents, but more generally it can be used for any information that can be represented as a labeled-tree. Each Web 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 correspond 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[4].

• **Navigation.** Navigation along the abstract tree is performed using path-expressions ([7,1]). A unique feature of HEL is that it comes with two ways

[3] like `http://us.imdb.com/Title?Ridicule+(1996)`

[4] A bachelor tag (aka empty tag) is a tag that does not require a closing tag, like `<IMG>` or `<BR>`.

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 [5] 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.

• **Index Variables and Conditions.** 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]` [6]) or an index range (like `[i:*]`, for all of them) that satisfies the condition. Conditions are introduced using index variables and `WHERE` clauses separated by `AND`. Disjunctions are not supported. 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 `"!"` [7] , 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 look at the structure of a movie entry

[5] list, since we care about the order of nodes.
[6] This is the default behavior.
[7] In the spirit of the Prolog cut.

from IMDb as shown in Figure 4, with the corresponding table structure [8] in Figure 5. To extract the genre of the movie, we need to find the table cell

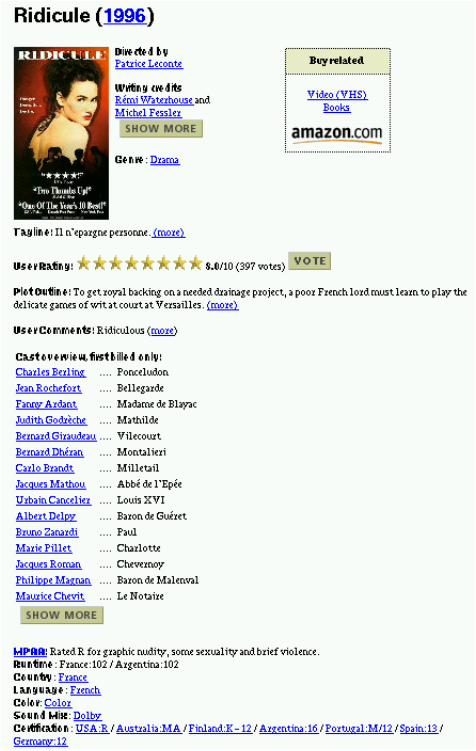

Fig. 4. The Web page

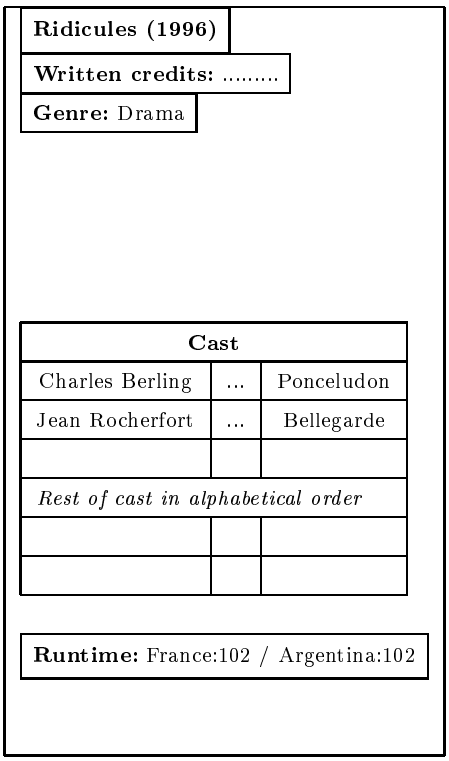

Fig. 5. The table structure

```
      html.body->td[i:0].a[*].txt
WHERE html.body->td[i].b[0].txt = "Genre";

      html.body->table[ii:0].tr[jj:*].td[0].txt, ...
WHERE html.body->table[ii].tr[0].td[0].txt =~ "Cast"
AND   html.body->table[ii].tr[jj].getNumberOf(td) = 3;
```

Fig. 6. The extraction rules.

of the document that starts with the string "Genre:" in boldface. Since we do not know the exact hierarchical structure of the nesting, we use the arrow operator with an index variable `i`. It also makes the extraction rule robust to any nesting modification. To extract the cast of the movie, we first need to identify the corresponding table. To do so, we introduce an index variable `ii` that gets resolved at runtime for the table that contains the string "Cast" in the first column of its first row. The extraction of the cast is generally straightforward except that for some movies the cast is split into a main cast and secondary cast [9]. In any case, we want to make sure that we do not extract the separator ("*Rest of cast in alphabetical order*"). To do so, we introduce a condition `jj` with a table row index variable that checks that the extracted

[8] Tables are useful in HTML to enforce text alignment.

[9] It is not the case for the movie presented in Figure 4.

row contains exactly 3 cells (`numberOf(td)`). The use of index variables makes extraction rules slightly more complex to write, but much more robust.

• **Regular expressions.** So far we have used 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 5, we might want to extract the title (`"Ridicule"`) itself and trim the year (`"1996"`).
To capture this level of details, our language comes with standard regular expressions à la Perl [28] 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 (the number of parenthesized sub-pattern binders indicates the number of items returned by the match.) 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 of the movie. `split` would be used when for instance the information is returned as a string with a delimiter. In the movie example, the runtime information could be extracted using two `split`s in cascade: `split ///, split /:/`. The string `"France:102/Argentina:102"` will be extracted as a list of pairs: `( ("France", "102"), ("Argentina", "102") )`.

• **Building Complex structures.**

```
movie = html.body(
          ->h1.txt, match/(.*?) [(]/
        # ->h1.txt, match/.*?[(]([0-9]+)[)]/
        # ->td[i:0].a[*].txt
        # ->table[ii:0].tr[jj:*].td[0].txt, match/(\S+)\s(.*)/
                )
where html.body->td[i].b[0].txt = "Genre"
and   html.body->table[ii].tr[0].td[0].txt =~ "Cast"
and   html.body->table[ii].tr[jj].getNumberOf(td) = 3;
```

Fig. 7. Building complex structures.

As pointed out previously, extraction should not be limited to isolated pieces of information but should be able to capture complex structures. From this perspective, HEL is different from XPath [26] – the XML navigation language – that can only return a set of nodes.
The HEL language therefore provides the fork operator `"#"` (like a record constructor) to build complex structures based on extraction rules. The meaning of the operator is somehow to follow multiple sub-paths at the same time and concatenate the results using a list semantics. 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. Instead of extracting pieces of information separately, we might want to get them as a whole. For a movie, we would write a slightly different extraction rule (Figure 7).

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 store the extracted information. Within W4F, information is stored in *Nested String Lists* (NSL), the datatype defined by:

$$\begin{aligned} NSL &= null \mid NSL' \\ NSL' &= String \mid list(NSL') \end{aligned}$$

It is important to note that items within a list can have different structures. The datatype has been chosen on purpose to be simple, anonymous – in the sense that the NSL does not have any label – 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 7 we can infer that for a `movie` the corresponding NSL will be a list of 4 items (3 top level forks). The first and second items are strings that represent respectively the title and year of the movie. The third item is a list of strings (`.a[*].txt`). The last is a list (`tr[jj:*]`) of pairs (`match` operator with two bindings) for the first name and last name of the actors in the movie.

NSLs are very low-level structures that can be manipulated via an API (list iterators and coercion operators). `NestedStringList` objects can be either `NSL_List` (list) or `NSL_String` (leaf). Lists can be iterated upon using `getItem`, while string values can be extracted from leaves using `getValue()`. An overview of the API is presented in Figure 8. A concrete application of the API will be presented in the next section (see Figure 9) as an illustration of the mapping.

```
abstract class NestedStringList
 |
 +----- class NSL_List
 |         int getLength()
 |         NestedStringList getItem( int i )
 |
 +----- class NSL_String
           String getValue()
```

Fig. 8. The NSL API.

5 Mapping information

As presented above, it is possible to manipulate the extracted information using the NSL API. But it is not very convenient and for a programming point of view we would prefer to handle Java types, where a movie title is represented as a character string and the year of release as an integer. W4F offers a mapping to Java objects and also ways to define mappings to some data formats such as XML, ASN.1, OIF [6], etc. In both cases, it is important to keep in mind that the structure generated out of mapping is constrained by the input NSL. Mappings can be seen as tree-transducers with limited restructuring capabilities.

• **Java mappings.** W4F offers a way to define mappings to Java. The user can specify a mapping to Java base types – and their array extensions – by simply indicating that the result of an extraction rule needs to be coerced to this type. The user can also specify a mapping to some user-defined Java types. In this case, the user needs to provide a valid Java class with a constructor that can convert the NSL input into an instance of this class.

The wrapper presented in Figure 3 takes advantage of Java mappings to extract the title as a `String`, the year as an `int`, the genres as a `String[]` and the cast as a `String[][]` (see Figure 9a).

If we decide to go for the complex structure presented in Figure 7, we would like the wrapper to export `movie` as a instance of a user-defined class `Movie`. We need to define class `Movie` with a valid constructor that takes the input NSL and manipulates it via the API to build the corresponding Java object. The API provides some methods to make it easy to build array types, using the Java reflection library. The content of the mapping section and the definition of the user-defined class are presented in Figure 9(a,b,c).

• **XML mappings with XML templates.** The ultimate goal of a wrapper is to export information according to a predefined interface. For Web sources, the XML format appears to be a good candidate to represent and exchange information. Having an automatic mapping from NSL to XML would be really convenient. From the last section, it is clear that such a mapping can be – quite – easily done by enriching each user-defined Java class with some methods to output XML. But as mentioned, such a mapping is neither generic nor declarative. In this section we explain how XML mappings can be defined inside W4F. Extensions to other data formats would be handled in a similar way.

An XML mapping expresses how to create XML elements out of NSLs. Before going further it is crucial to understand that the *shape* of XML elements we can generate is constrained by the structure of the NSL itself. XML mappings can be seen as tree transducers that take an NSL as an input and output an XML tree.

An XML mapping is described via declarative rules called *templates*. Tem-

(a) Mapping to Java base types

```
SCHEMA
{
String title;
int year;
String[] genres;
String[][] cast;
}
```

(b) Mapping to a user-defined Java class

```
SCHEMA
{
Movie movie;



}
```

(c) Implementing the user-defined mapping

```
public class Movie
{
  String title;
  int year;
  String[] genre;
  Actor[] cast;

  public Movie( NestedStringList nsl )
  {
    NSL_List list = (NSL_List) nsl;
    title = ((NSL_String) list.getItem(0)).getValue();
    year  = Integer.parseInt( ((NSL_String) list.getItem(1)).getValue() );
    genre = (String[]) NSL.toObjectArray( list.getItem(2) );
    cast  = (Actor[])  NSL.toObjectArray( list.getItem(3), Actor );
  }
}

public class Actor
{
  String firstName, lastName;

  public Actor( NestedStringList nsl )
  {
    NSL_List list = (NSL_List) nsl;
    firstName = ((NSL_String) list.getItem(0)).getValue();
    lastName  = ((NSL_String) list.getItem(1)).getValue();
  }
}
```

Fig. 9. Java Mappings.

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

```
<!ELEMENT Movie (Title,Year,Genres,Cast)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT Genres (Genre)*>
<!ELEMENT Genre (#PCDATA)>
<!ELEMENT Cast (Actor)*>
<!ELEMENT Actor (FirstName,LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

```
<Movie>
  <Title>Ridicules</Title>
  <Year>1996</Year>
  <Genres>
    <Genre>Drama</Genre>
  </Genres>
  <Cast>
    <Actor>
      <FirstName>Charles</FirstName>
      <LastName>Berling</LastName>
    </Actor>
    <Actor>
      <FirstName>Jean</FirstName>
      <LastName>Rochefort</LastName>
    </Actor>
...
  </Cast>
</Movie>
```

Fig. 10. The template, the DTD and document

plates are nested structures composed of *leaves*, *lists* and *records* and are defined using the XML Template language. Templates always start with a `"."` because we assume that the generated XML elements will be inserted as part of an already existing XML document. Before we explain in details the constructs of the language, let us consider a simple XML mapping for the movie example. Figure 10 presents a template, the corresponding DTD and the XML document it would produce when applied to the movie "Ridicules". The first thing to notice is that the structure of the template is closely related to the structure of the extraction rule. The mapping will create a `<Movie>` element with four sub-elements: `<Title>`, `<Year>`, `<Genres>` and `<Cast>`. `<Title>` and `<Year>` are string-valued (`PCDATA`). `<Genres>` is a repetition of zero or more `<Genre>` sub-elements (string-valued). `<Cast>` is a repetition of `<Actor>` sub-elements, where an actor contains two string-valued sub-elements (`<FirsName>` and `<LastName>`). The details of the template language are defined below. The semantics can be found in the appendices, for both the translation of a template into a DTD (B) and for the generation of an XML document out of an NSL, for a given template (C).

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

Fig. 11. The XML template BNF.

We detail next each type of template. In figures 12 and 13, we present for each type of template the various constructs. Each construct is described on three rows: the first row is the template construct; the second row is the corresponding DTD element declaration; and the third row is an instance of an XML element produced by the mapping.

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. The sequence __ in the examples below simply means "anything".

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.

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

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

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

Fig. 12. Leaf templates.

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

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

Fig. 13. Record (left) and List (right) templates

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. The construction of the DTD is straightforward from the specification itself. The semantics of the translation is described in the appendix.

Two important remarks about the mapping are worth mentioning.
First, 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.
Second, the template language can only create a subset of all the possible DTDs. For instance it is not possible to produce DTDs with a content-model that makes use of `+`, `?` and `|`.

6 Visual Support

The last component of the system we present is the suite of visual tools 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.

• **Support for Writing Extraction Rules.** The role of the extraction wizard (see Figure 14) 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 from Figure 4.

Now, when the user points to "Ridicule", the corresponding text element gets high-lighted (the user can identify information boundaries enforced by the HTML tagging) and the canonical [10] extraction rule pops-up. The "magic"

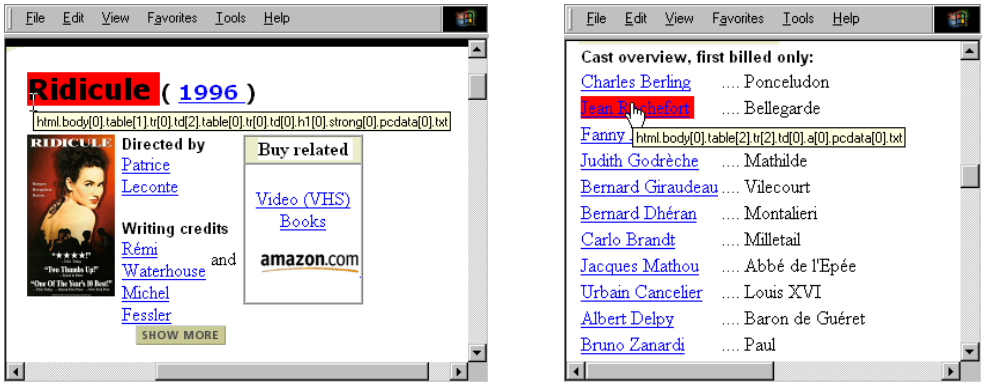

Fig. 14. The extraction wizard in action on the movie page.

behind it takes advantage of our DOM-centric approach: when the page is fed into the parser, each text chunk (i.e. `PCDATA`) gets annotated with its corresponding canonical path in the document tree. As an illustration, the annotation of a tree for the movie example looks like the following:

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

```
...<H1><STRONG>Ridicule</STRONG></H1>...
```

gets annotated as

```
...<H1><STRONG>
<SPAN ID="html.body[0].table[1].tr[0].table[0].tr[0].h1[0].strong[0].pcdata[0].txt">
Ridicule
</SPAN>
</STRONG></H1>...
```

This systematic annotation strategy carries some restrictions. First, the path produced is the *canonical path*: it does not use all the powerful constructs of the HEL language like `"->"`, index ranges, conditions or regular expressions. Second, the annotation is done on a per element basis. In the example of Figure 14, it would be convenient to be able to point to all the items from the cast list – not just one – and get the extraction rule for the entire cast.

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 (see Figure 3). For the title, in the actual wrapper we have short-circuited the canonical path using the `"->"` operator. For the cast, the actual extraction rule has a similar structure as the one returned by the wizard, where index constants have been replaced by index variables.
In any case, the extraction wizard always provide some useful *local* information.

• **Visualizing the Wrapper.** Another useful interface permits to test and refine the wrapper interactively before deployment. Figure 15 shows the wizard which visualizes the 3-layer architecture of the wrapper. In the first layer, the user inputs the location of the Web source and the retrieval method (a `GET` by default). The second layer displays the extraction rule – expressed in the HEL language – to be applied on the retrieved HTML page. In this example, the rule tries to extract the title, the year and the cast of the movie. The third layer presents the XML mapping to be applied to the information extracted. The last layer displays the extracted NSL on the left and on the right the XML document produced out of it.
The wizard is especially useful because extraction rules can be refined interactively.

7 Information Integration Using W4F: Building a TV Agent

After exposing the technical details of the toolkit, we now show how it can be used to solve a typical information integration problem involving Web information sources. We first detail a motivating scenario, before we explain how to build the TV Agent using Web wrappers and XML-based integration tools.

• **The scenario.** It is 7PM and you are about to go back home. But before

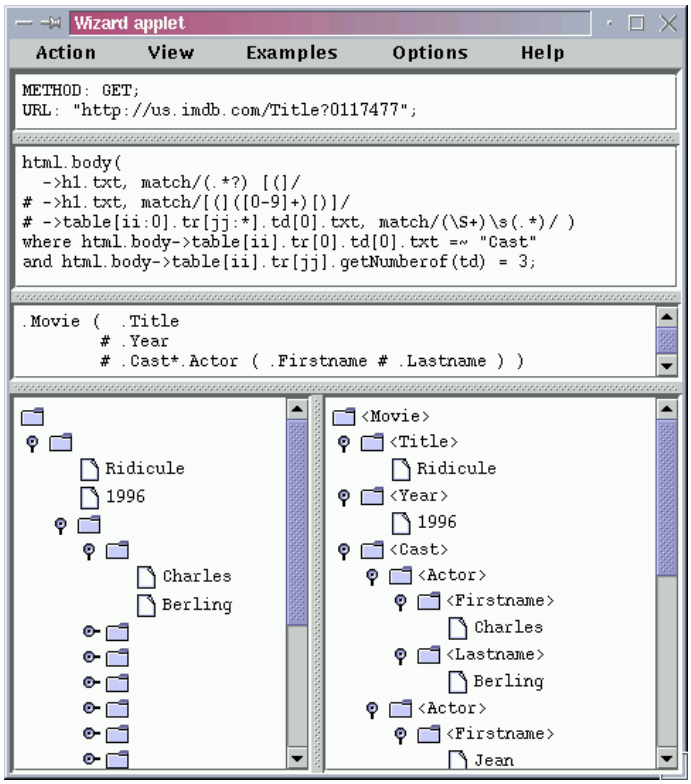

Fig. 15. A Visual View of the Wrapper.

leaving you would like to know if there is a good movie around 9PM tonight. As a Web savvy, you go to get the TV listing on your favorite Web site.
For a given time frame, the Web site displays the list of programs available for each channel. A table cell defines the beginning and the ending of a program. Types of programs are identified by a color code. A screen-shot of the TV program is presented below in Figure 16. As a human being processing the information, you first decide to ignore channels that you do not pay for. Then, for each movie, you want to gather some detailed information. The TV guide offers a brief description of the movie, but unfortunately, this is not enough for your needs. Therefore, you decide to grab some extra details from the Internet Movie Database. You connect to the web site and type the title in the input form and get back the movie description with all the details you need: genre, cast, director, language, country, rating, etc. (see Figure 4). If you are a thorough movie fan, you will also go to one or more movie review Web sites to gather some critics about this specific movie. And you would have to repeat the same process, for every movie of the listing. What you would really like is to have a personal assistant that would know your profile and ask for your today's requirements: it would perform the entire process for you and would notify you with a brief report.
In the following, we show how this problem can be tackled using W4F and we present a concrete solution.

• **The wrappers.** For the TV listing, we need to capture the table structure.

Fig. 16. TV listings Web page (from http://tv.yahoo.com).

By looking[11] at Figure 16, we see that the TV listings consist of tables (1 table per chunk of 10 channels). The first table is just used for navigation. For each table, the first row displays the time frames. The other rows represents the programs, one row per channel. The first and the last column contain the name of the channel. The columns in between contain the name of the program. The size of the column indicates the duration of the program (1 column unit corresponds to 30 minutes).

The problem is now to extract enough structure in order to be able to reconstruct the entire TV listings inside our application. The extraction rule for the TV program is presented in Figure 17. The condition is used to make sure that the last column is always thrown away. The last column happens to use bold-face characters, hence the `font[0].numberOf(b) == 0` predicate.

```
html.body.table[1-].tr[1-](
              .td[0].txt                        // channel name
            # .td[i:*] ( .txt                   // program name
                       # .getAttr(colspan)      // duration
                       # .getAttr(bgcolor)      // program genre
                       )  )
WHERE html.body.table[1-].tr[1-].td[i].font[0].numberOf(b) == 0;
```

Fig. 17. Extraction rule for TV listing.

The data-structure extracted from the page consists of a list of channel chunks. A channel chunk is a list of channels. A channel is a list of two items. The first item is the name of the channel; the second item is a list three items: program name, duration and program genre. It is important to remark that

[11] The extraction wizard turns out to be extremely useful to help discover the structure of Web pages.

there is no magic here. The extracted structure is not enough to reconstruct the information available from the HTML page. First we need to remember the starting time of the TV grid. We could extract it from the page but it is better to assume that we know it since we access the TV listing by providing this piece of data as an input value. Second, we need to remember some details about the TV listing such as the fact that a column represents 30 minutes as well as the color code for TV programs. We will assume that these details are taken care of by the rest of the application.
We also need to design a similar wrapper for the movie source. The details of the wrapper for the Internet Movie Database have already been presented in Section 3 and appear in Figure 3.

• **Integration using XML and related tools.** A good and elegant way to perform the integration is to define an XML representation for all the entities involved in the application and define some mediation at the level of XML. For this task, we will use XML-QL [9], a query language proposed to query XML documents. Some interesting features of the XML-QL query language are: it is declarative; it is "relational complete"; it is compositional (takes one or more XML documents and generates a new XML document); and it can support both ordered and unordered views on an XML document.

First we need to map the information extracted by our wrappers into some XML structure. Using W4F, we can define a mapping (Figure 18) for the information about the TV listings. A piece of the XML document generated is presented in Figure 19, with its DTD.

```
TV_Listing.Channels**.Channel( .ID^
                              # .ProgramList*.Program ( .Title
                                                      # .Duration^
                                                      # .Code^     ))
```

Fig. 18. XML mapping for the TV listings.

```
<TV_Listing>
  <Channels>
    <Channel ID="SUNDAE 1">
      <ProgramList>
        <Program Duration="4" Code="#b0e0e6">
          <Title>Ridicule (1996) *** (R)</Title>
        </Program>
        <Program Duration="2" Code="#b0e0e6">
          <Title>Thieves (1996) *** (R)</Title>
        </Program>
      </ProgramList>
    </Channel>
    <Channel ID="FOX 2">
      <ProgramList>
        <Program Duration="2" Code="#b0e0e6"> ..
```

```
<!ELEMENT TV_Listing (Channels)>
<!ELEMENT Channels (Channel)*>
<!ELEMENT Channel (ProgramList)>
<!ATTLIST Channel ID CDATA #IMPLIED>
<!ELEMENT ProgramList (Program)*>
<!ELEMENT Program (Title)>
<!ATTLIST Program
   Duration CDATA #IMPLIED
   Code CDATA #IMPLIED>
<!ELEMENT Title (#PCDATA)>
```

Fig. 19. The XML document and its DTD

As mentioned above, we need to enrich a little bit the XML structure to capture all the information from the original document. We have to walk the tree and replace duration by the actual starting and ending time of the movie,

based on the global information we know about the TV Guide. We then do the same for movies. The details of the mapping have already been presented in Section 5.

• **Putting everything together in a XML-QL query.** We can now express integration as an XML-QL [12] query. More details about the semantics of XML-QL can be found in [9].

The query appears in Figure 20 with its output in Figure 21. The `WHERE` clause consists of 3 tasks: (1) retrieving the TV_Listing as an XML document and create some bindings for `$channel_id`, `$start`, `$end` and `$t`, based on the structure of the document; (2) retrieving each movie according to binding `$t` and creating some bindings for `$title`, `$genre` and `$country`, based on the structure of the document; (3) enforcing some constraints for the various bindings. The result of the `WHERE` clause can be seen as a relation with a column for each variable name and a row entry for each binding. The `CONSTRUCT` clause simply consumes the bindings and generates an XML document accordingly. The result consists of one unique XML document.

```
CONSTRUCT
<CHOICE START=$start END=$end CHANNEL=$channel_id>
  <MOVIE>
    <TITLE>$title</>
    <YEAR>$year</>
    <Country>$country</>
   </>
</>
WHERE
<TV_Listing.Channels.Channel ID=$channel_id>
    <ProgramList>
      <Program START=$start END=$end>
        <Title>$t</>
      </>
    </>
</> in URL:TV_Listing( date, time ),
$channel_id = "Sundance"
<Movie>
  <Title>$title</>
  <Year>$year</>
  <Genres.Genre>$genre</>
  <Country>$country</>
</> in URL:IMDB_Movie( $t ),
$year < 1990,
$genre != "Sci-Fi",
$country = "France"
```

Fig. 20. The XML-QL query.

8 Experience with W4F

In this section we describe some other applications that have been (or could be) designed using W4F and mention some strengths and weaknesses of the

[12] The query assumes an extension of XML-QL to handle dependent joins.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CHOICE START="9PM" END="11PM" CHANNEL="SUNDAE1">
  <MOVIE>
    <TITLE>Ridicules</TITLE>
    <YEAR>1996</YEAR>
    <COUNTRY>France</COUNTRY>
  </MOVIE>
</CHOICE>
<CHOICE START="11PM" END="?" CHANNEL="SUNDAE1">
  <MOVIE>
    <TITLE>Voleurs, Les</TITLE>
    <YEAR>1996</YEAR>
    <COUNTRY>France</COUNTRY>
  </MOVIE>
</CHOICE>
```

Fig. 21. The XML document that represents the result of the query.

toolkit based on our own practical experience and some feedback from research and corporate users.

8.1 Examples of applications developed using W4F

• **Conversion Tools.** Lightweight wrappers can be used to convert HTML data into anything. The toolkit already offers a default mapping to Java objects. It also offers a declarative specification to map HTML into XML as presented in the previous section. New mappings can be easily added using Java classes.

W4F has been particularly successful to write XML gateways that offer on-the-fly conversion from HTML to XML. Thanks to such gateways, Web information sources can be looked at *through XML glasses* [23] for structured processing and HTML pages can be *recycled* [24] as XML documents.

• **Data migration.** Lightweight wrappers are also suitable for migrating Web content into a data warehouse architecture. Wrappers handle at the same time extraction, cleaning and restructuring. They can be used for instance to migrate the content of databases available on the Web into corporate repositories, virtual or materialized.
Data migration can also include the building of large knowledge bases populated with information gleaned on the Web. For instance, On2Broker [10] and SIMS [20] which offer a query interface to the CIA World Factbook could make use of the wrapper presented in [24] instead of hand-crafted ones.

• **Information gathering agents.** With minimal effort, using W4F it is possible to write a meta-search engine on top of AltaVista, HotBot and Excite, a shopping agent like Jango (http://www.jango.com) or a portfolio manager. The major benefit of W4F for this domain is that it permits to make the content of any Web information source available to the application.

• **Value-added network / portal development.** Value added networks can leverage the value of single Web sources by making them work together through lightweight wrappers. Portals offer an entry point to information from

various resources. In both cases, W4F wrappers can make the integration of new resources quick and easy.

8.2 Other issues

• **Expressiveness of the language.** One major strength of the toolkit is the expressiveness of the extraction language, especially the use of index variables and complex structure. By using index variables, it is possible to postpone until runtime the decision about which value to pick. This is useful when the structure of the page depends on the nature of the query. When a relational database outputs the results of a user query in HTML, the ordering on the columns depends on the structure of the query. Using index variables, the column can be defined based on its name, not its position. Complex structures are also valuable because they permit to take advantage of locality. Instead of returning one piece of information, the fork construct permits to identify the information and return surrounding pieces in a structured way.

The main limitations we have encountered involves text content that uses tags as standalone delimiters rather than containers. For instance, `TABLE` completely defines a region (contained between `<TABLE>` and `</TABLE>` while `<H1></H1>` just defines the beginning of a region. The structure of some HTML pages is implicitly defined by patterns of standalone delimiters and it is sometimes difficult to write extraction rules elegantly. The `"!"` operator turns out to be a way to solve this problem.

Fortunately, W4F usually offers more than one way to tackle such problems. When the structured navigation is not suitable, it is always possible to identify a larger region that contains the information, get the corresponding HTML source (using the `.src` property) and then apply NSL operators.

• **More semantics.** For some domain specific applications, some users have asked for extraction functions with more semantics, in order to be able to extract dates, invoice numbers, DNA sequences, etc. These requirements fits perfectly in our framework through user-defined functions. Like for the previous point, the structured navigation can be used as far as it can, and the rest of the processing is handed to some specific Java code.

Semantics also means to be able to define some classes of tags. For instance `<EM>`, `<I>` and `<B>` have a similar purposes and our extraction language should take advantage of it.

• **Robustness of extraction rules.** A big concern when dealing with wrappers is not the authoring of wrappers but their maintenance. As reported in [16], the lifetime of a wrapper is around one month. Our solution is to make the authoring fast which means that maintenance often means rewriting the wrapper. The trade-off is between robustness and simplicity of the wrapper. We do not have empirical evidence, but the use of an HTML specific extraction language combined with some powerful constructs makes our wrappers

quite robusts.

• **Performance.** For most processing, the first bottleneck is the network connection. When processing local files (which is often the case when W4F wrappers are used in an intranet environment), the second bottleneck is the use of DOM. DOM requires the entire document to be built in memory. For the large majority of our applications where the size of the documents is small (a few KBytes), the bottleneck has been network delays. The cleaning of HTML is also sometimes very expensive, depending on the ill-formedness of the document.
The evaluation of our extraction rules already uses some optimization technique that avoid multiple navigations of the tree. Some improvements we are looking at concern the cleaning and pruning of the original document. In most cases, we know at compile time that some attributes or elements are not going to be used by the extracting rule: therefore there is no need to include them in the DOM tree.

• **Other issues.** Not surprisingly, a main limitation of the framework concerns the retrieval of HTML documents. In some cases, the only way to get to the HTML document is through frames, JavaScript interaction and cookies, none of which are not yet supported by W4F.

• **Empirical evidence.** As of this writing, we have no scientific empirical evidence about the benefits of our approach in terms of user-friendliness. The only argument we can make is based on the size of the W4F wrappers compared to other frameworks and the number of wrappers authored by people.

9 Related work

In this section, we compare our approach to others, with respect to various components of the system.

• **Retrieval.** Frameworks like WebL [13] and WIDL [3] offer some advanced features for retrieving Web pages. 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.

In W4F, the retrieval is described declaratively, but issues like recovery or the exact semantics of the retrieval are not addressed [13] , in order to keep wrappers as simple as possible.

• **Extraction.** An important aspect of extraction deals with how the document is represented. On the one hand, a Web document can be viewed as a flow of tokens that can be processed through regular expressions (Tsimmis [12]), ex-

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

pressive grammars (Araneus [18], SIMS [19,20]), or text algebras (WebL [13]). 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 ([11], [3], XQL [25]) or a similar semi-structured data-model (XML-QL [9], Web-OQL [4]). However, navigation along this explicit structure is sometimes restricted to the hierarchy itself and cannot capture finer granularity information.

W4F tries to combine both approaches by allowing tree navigation and regular expressions. 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. Moreover, it allows to extract complex constructs and not just atomic nodes – or flat collection of nodes – from the DOM tree, in order to capture the implicit structure of the information of the document. To the best of our knowledge, HEL captures all the features of the other DOM-based languages.

• **Mapping.** Wrappers are in charge of providing a structured access to the extracted information. For Web-OQL [4], a Web document is an OQL instance from the beginning. In Tsimmis [12] the extracted information is converted into the OEM format. [11] offers CORBA-like interfaces. YAT [8] offers a very expressive rule-based framework (fully declarative) to express mapping as generic tree transformations. Clearly our mappings are not as expressive as the ones offered by YAT for instance, but our framework is flexible enough to export its structures for further processing by other tools.

• **XML.** Our tackling of XML is different from the one of XML-QL [9] based on patterns and explicit constructs because we derive it from our extraction process that handles HTML pages with no explicit structure. For the same reason, our XML templates are more restrictive than XWrap [17]. As pointed previously, the range of XML documents we can create is very limited, due the choice of our template language. We think that it is important to offer an easy way to specify *one* mapping, knowing that it is always possible to transform the generated XML document(s) using other tools.

• **Wrapper Engineering Strategies.** The manual generation of a wrapper often involves the writing of ad-hoc code ([12] and [18]). Web-OQL [4] 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 wrappers. In WIDL [3], the entire structure understood by the system is presented to the user who has to pick what he wants. In [2] and [17], the user is presented a dual view of the document with its layout and its corresponding tree. SIMS [20] and LiveAgent [14] offer a *demonstration-oriented* interface where the user shows the system what information to extract. In [16] and [15], Kushmerick uses machine-learning techniques to generate wrappers automatically.

Extraction is defined according to some classes of wrappers that need to be trained with some examples, under human supervision. Machine-learning is used at the level of tokens and has no real understanding of the document structure, which makes wrappers more generic (for any text content) but also less robust. These techniques are really promising but only support a subset of our extraction primitives.

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.

• **Visual support.** Like [3,2,17,20], W4F offers some visual support to help the generation of wrappers. However, the level of visual support is unable to match the expressivity of our extraction language, which is not a concern for the other approaches.
Like XWrap [17], we offer a wysiwyg support where the user selects the information to extract from the original document.

• **Mediation.** Finally 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 [12], Garlic [21], etc.

10 Conclusion and future work

We have presented the World Wide Web Wrapper Factory, a toolkit for generating wrappers for Web information sources. 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 various data-formats such as XML; (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.

We have demonstrated that our Web wrappers are useful ingredients for the development of Web applications. They permit access to data without requiring the the Web source to be changed. They interoperate with other integration components via mapping to Java or XML. Their simplicity makes it quite easy to cope with the versatility of Web sources. They are scalable because they are easy to deploy on a wide range of platforms and require little resources. Finally, they use Web standards like HTTP and XML and can be directly integrated into bigger information systems. We have also presented some types of applications that have already or could benefit from the use of such wrapper methodology.

There are some evident directions for future work. First, it is important to offer better support to wrapper authors. Crafting extraction rules still requires

significant expertise. We need to investigate the use of machine-learning techniques to define robust shortcuts for complicated extraction paths. Second, we need to enrich our extraction language. For instance 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. Another interesting issue is how to offer an extraction language that combines structured navigation using path expressions with text-algebra manipulation as in [13]. Third, we think that maintenance is a crucial aspect that has to be addressed properly. Among other things, it involves defining some heuristics to identify when a Web source has been changed (not in terms of content, but of layout), and being able to simulate changes to see how robust extraction rules are. Finally, we would like to migrate the wrapper framework from a database oriented to an agent-based environment, where tasks are more collaborative and goal-oriented.

W4F has been successfully used to generate a large variety of Web wrappers for information sources and build Web applications. The toolkit [22] can be downloaded from the Penn Database Research Group Web site[14]. On-line examples of W4F applications (including the wrappers presented in this article) 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] Fabien Azavant and Arnaud Sahuguet. *W4F User Manual.* Tropea Inc., 2000. Availalble from `http://www.tropea-inc.com`.

[6] R.G.G. Cattell, editor. *Object Database Standard ODMG 2.0.* Morgan Kaufmann, 1997.

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

[8] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In *Proc. SIGMOD Conference*, Seattle, 1998.

[14] http://db.cis.upenn.edu/W4F

[9] 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.

[10] Dieter Fensel and al. On2broker: Semantic-Based Access to Information Sources at the WWW. In *Workshop on Intelligent Information Integration (III99)*, August 1999.

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

[12] 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.

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

[14] Bruce Krulwich. Automating the Internet: Agents as User Surrogates. *IEEE Internet Computing*, 1997.

[15] Nicholas Kushmerick. Gleaning the Web. *IEEE Intelligent Systems*, 14(2), 1999.

[16] Nicholas Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2), 2000.

[17] Ling Liu, Calton Pu, Wei Han, David Buttler, and Wei Tang. An XML-based Wrapper Generator for Web Information Extraction. In *ACM SIGMOD International Conference*, June 1999.

[18] 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.

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

[20] 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.

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

[22] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy Web data-sources using W4F. In *International Conference on Very Large Databases (VLDB)*, 1999.

[23] Arnaud Sahuguet and Fabien Azavant. Looking at the Web through XML glasses. In *CoopIs*, 1999.

[24] Arnaud Sahuguet and Fabien Azavant. Web Ecology: Recycling HTML pages as XML documents using W4F. In *WebDB*, 1999.

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

[26] W3C. XML Path Language (XPath) 1.0. W3C Recommendation 16 November 1999. Available from http://http://www.w3.org/TR/xpath.

[27] Philip Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, Philadelphia, December 1999.

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

[29] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

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

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

A Semantics of the HEL language

To describe the semantics of the HEL extraction language, we will use a data-model similar to the one presented by Phil Wadler in [27] for the semantics of XSL-T.

An HTML document is represented as a collection of *Node*s that define a tree structure. Our data-model does not make a special case out of references and treats `ID` and `IDREF`s as regular attributes.

Every node has a name and can be of one the following kinds: *element* (for HTML elements), *attribute* (for HTML attributes) or *text* (for PCDATA). Every node has a value: for text nodes, it corresponds to the text content; for attribute nodes, to the attribute value; and for element nodes, to the concatenation of the values of its children nodes (visited recursively in order).

For each kind, we assume the existence of a boolean function of type *Node* $\rightarrow$ *bool* that tests the kind of a node. The tree structure of the document is defined by the *parent*, *children* and *attributes* relationships between nodes. The content of the document is defined by node values. These relationships and values are described by the following functions, where Set_1 represents sets with 0 or 1 element:

We also define a total order (noted $\leq_{doc}$) on document nodes. The ordering corresponds to a depth-first traversal of the document tree.

$$
\begin{array}{lcl}
parent & : & Node \rightarrow Set_1(Node) \\
children & : & Node \rightarrow list(Node) \\
attributes & : & Node \rightarrow list(Node) \\
root & : & Node \rightarrow Node \\
tagName & : & Node \rightarrow String \text{ (tag name or attribute name)} \\
value & : & Node \rightarrow String
\end{array}
$$

It is worth noting that for a given element, its attributes (if any) are not part of its children, but are reached via the *attributes* function. We will also define the function *successors: Node →list(Node)* that returns the list of nodes that are found after a given node (in the sense of the $\leq_{doc}$ order).

Figure A.1 is an example of an HTML document and its encoding in this data-model. The document is encoded as a collection of nodes ($o_1..o_8$) and some functions. For clarity we omit the value of nodes that have children. The *Root* function is constant and returns o_1.

```
<HTML>
<HEAD>
<TITLE>Example</TITLE>
</HEAD>
<BODY>
<H1 ALIGN='center'>Welcome</H1>
</BODY>
</HTML>
```

node	name	kind	parent	child.	attr.	value$^{(*)}$
o_1	HTML	Elem	$\emptyset$	$[o_2, o_5]$	[ ]	
o_2	HEAD	Elem	o_1	$[o_3]$	[ ]	
o_3	TITLE	Elem	o_2	$[o_4]$	[ ]	
o_4	PCDATA	Text	o_3	[ ]	[ ]	Example
o_5	BODY	Elem	o_1	$[o_6]$	[ ]	
o_6	H1	Text	o_5	$[o_7, o_8]$	o_6	
o_7	ALIGN	Attr	o_6	[ ]	[ ]	center
o_8	PCDATA	Text	o_6	[ ]	[ ]	Welcome

Fig. A.1. Encoding

We now give semantics to HEL expressions by specifying how they map instances of HTML data into NSL structures. An HEL expression will operate on this data-model to return an NSL. NSL structures are defined by the following:

$$
\begin{array}{lcl}
NSL & = & null \mid NSL' \\
NSL' & = & String \mid list(NSL')
\end{array}
$$

In order to describe the semantics we also need to introduce another data-type called *Nested Node List (NNL)* and defined as:

$$
\begin{array}{lcl}
NNL & = & null \mid NNL' \\
NNL' & = & Node \mid list(NNL')
\end{array}
$$

We will use :: as the list constructor for list construction and list pattern matching and @ for list concatenation. Lists will be represented between [ ]. It is important to note that these datatypes treat *null* and [ ] (the empty list) as different. When there is an ambiguity between both datatypes, we will write $[\,]^{NNL}$ and $[\,]^{NSL}$.

The use of NSLs has already been motivated in a previous section. As for NNLs, they are intermediate structures used when evaluating HEL expressions. It is important to understand that *NNL are not used to represent the structure of the document, but to represent the structured state of the HEL navigation on the HTML document.* Should we use unstructured states instead of NNLs, we would not be able to construct complex nested structures. For instance, using NNLs, `html->tr[*].td[*]` can be represented as a list of list of nodes instead of a flat list of nodes. NNLs are used to keep track of the navigation inside the document, based on the path components of the extraction rule.

We now introduce an abstract syntax for a subset of our extraction language (Figure A.2). This subset describes condition-free extraction *paths*, with node operators (*nodeOp*) and NSL operators (*nslOp*). The issues related to conditions will be dealt with later on in the section.

rule	=	*name* `"="` `"html"` *path* `";"`
path	=	`"."` *tag* `"["` *index range* `"]"` *path*
	|	`"->"` *tag* `"["` *index range* `"]"` *path*
	|	*path1* `"#"` *path2*
	|	*op*
op	=	*nodeOp* | *nodeOp* `","` *nslOps*
nodeOp	=	`".txt"` | `".src"`
	|	`".getAttr("` *attrName* `")"` | `".getNumberOf("` *tag* `")"`
nslOps	=	*nslOp* | *nslOp* `","` *nslOps*
nslOp	=	`"regex("` *regex* `")"` | `"split("` *regex* `")"` | *user function*
tag	=	*string*
name	=	*string*
attrName	=	*string*
regex	=	*string*
index range	=	*e_range* | *i_range* | *i_range* `","` *index range*
e_range	=	`"*"` | *integer* `"-"`
i_range	=	*integer* `"-"` *integer* | *integer*

Fig. A.2. Simplified grammar of the extraction language

The semantics of the extraction language is defined via 3 *curried*[15] functions:

[15] A function of N arguments can be considered as a function of one argument which returns another function of N-1 arguments.

– $\mathbb{E}[\![\ ]\!]$: *path* →*NNL* →*NSL* represents the evaluation of an extraction path on a document. Given a path and a NNL, it will return an NSL. An extraction rule defined by path p will be evaluated by calling $\mathbb{E}[\![\ ]\!]$ on path p with the NNL that consists of the root node of the HTML document.
– $\mathbb{E}_{\mathrm{N}}[\![\ ]\!]$: *nodeOp* →*NNL* →*NSL* represents the application of node operators to extract node information and convert it to string values (NSL).
– $\mathbb{E}_{\mathrm{S}}[\![\ ]\!]$: *nslOp* →*NSL* →*NSL* represents the application of NSL operators on *Nested String Lists*, like regular expression operators or user-defined functions.

• **HEL navigation and evaluation.** HEL navigation is tricky because it constructs complex nested structures and not flat sets. Complex structures are created – as mentioned before – using index ranges or forks.

For index ranges, we need to distinguish between singleton index ranges (that expect a single element) and multiple index ranges. It is crucial to understand that `html->a[0]` should return a single node (or null) while `html->a[*]` should always return a list. In the case of a document with only one `<A>` tag, the first extraction should return a single node while the second should return a list with one single element.

In our semantics, we capture both cases in a uniform way by abstracting on index ranges. α represents an index range that can be an integer, an interval or a union of them. Index ranges can also be infinite like "$*$" or "$2-$".

To make things simpler, we normalize index ranges into *range lists*, which are list of positive integers in increasing order, optionally terminated by the "$*$" symbol. We define the function $\mathbb{N}$*: index range* → *range list*. Normalization simply consists of expanding intervals and getting rid of the "$-$" symbol.

The normalization is defined as follows:

$$
\begin{array}{lcl}
\mathbb{N}(\text{"}i\text{"}) & = & [i] \\
\mathbb{N}(\text{"}i-j\text{"}) & = & [i, i+1, .., j] \\
\mathbb{N}(\text{"}i-\text{"}) & = & [i, *] \\
\mathbb{N}(\text{"}*\text{"}) & = & [*] \\
\mathbb{N}(\text{"}i, \textit{index range}\text{"}) & = & i::\mathbb{N}(\textit{index range}) \\
\mathbb{N}(\text{"}i-j, \textit{index range}\text{"}) & = & [i, i+1, .., j]@\mathbb{N}(\textit{index range})
\end{array}
$$

Fig. A.3. Normalization of index ranges

Depending on the nature of the range list (i.e. index range), the result of applying α to an NNL and noted $\mathbb{R}(nnl, \alpha)$ is going to be different. $\mathbb{R}$ is a function with signature: $NNL \rightarrow$ *range list* $\rightarrow NNL$.

The notation means that the index range must be applied to the list in the following sense:

– if α is an integer, applying it to a list means to return the αth element of

the list if it exist, or null.
- if α is a range, applying it to a list means to extract the corresponding elements and return them as a list.

More precisely, the semantics of index ranges is defined in Figure A.4, where we assume the existence of the *dec* function that takes a (strictly positive) range list and decrements each element by one.

$$
\begin{array}{lcl}
\mathbb{R}(nnl, [\,]) & = & [\,] \\
\mathbb{R}(nnl, [*]) & = & nnl \\
\mathbb{R}(n :: nnl, 0 :: rangeList) & = & n :: \mathbb{R}(nnl, rangeList) \\
\mathbb{R}(n :: nnl, i :: rangeList) & = & \mathbb{R}(nnl, (i-1) :: dec(rangeList)) \text{ for } i > 0 \\
\mathbb{R}([\,], i :: rangeList) & = & \text{FAIL}
\end{array}
$$

Fig. A.4. Index Range Semantics

For the last case, we will throw an *exception* that will be handled at the level of NSLs. The issue is the same as dealing with division by zero when defining the semantics of arithmetic expressions.

At the level of NSL, in the case where the node to which we try to apply the path is null, the result depends on the nature of the path. If the expected result of the path should be a single-valued NSL, the result is null. If the the expected result of the path should be list, the result is the empty list. We define the function *isSingleValued*: $path \rightarrow bool$ that returns true is the left most index range in the path is single valued.

The semantics is presented in Figure A.5.

$$
\begin{array}{lcl}
\mathbb{E}[\![\, path \,]\!]\,\text{FAIL} & = & null^{NSL} \text{ (handles the exception)} \\
\mathbb{E}[\![\, path \,]\!]\, null^{NNL} & = & \text{if } isSingleValued(path) \text{ then } null^{NSL} \text{ else } [\,]^{NSL} \\
\mathbb{E}[\![\, path \,]\!]\,[\,]^{NNL} & = & \text{if } isSingleValued(path) \text{ then } null^{NSL} \text{ else } [\,]^{NSL} \\
\mathbb{E}[\![\, p_1 \texttt{\#} p_2 \,]\!]\, node & = & \mathbb{E}[\![\, p_1 \,]\!]\, node :: \mathbb{E}[\![\, p_2 \,]\!]\, node \\
\mathbb{E}[\![\, \texttt{.}\text{tag}[\alpha]\ path \,]\!]\, node & = & \\
 & & \mathbb{E}[\![\, path \,]\!]\, \mathbb{R}([x \mid x \leftarrow children(node) \wedge name(x) = tag], \mathbb{N}(\alpha)) \\
\mathbb{E}[\![\, \texttt{->}\text{tag}[\alpha]\ path \,]\!]\, node & = & \\
 & & \mathbb{E}[\![\, path \,]\!]\, \mathbb{R}([x \mid x \leftarrow successor(node) \wedge name(x) = tag], \mathbb{N}(\alpha)) \\
\mathbb{E}[\![\, NodeOp \,]\!]\, node & = & \mathbb{E}_{\mathrm{N}}[\![\, NodeOp \,]\!]\, node \\
\mathbb{E}[\![\, path \,]\!]\, (node :: l) & = & (\mathbb{E}[\![\, path \,]\!]\, node) :: (\mathbb{E}[\![\, path \,]\!]\, l)
\end{array}
$$

Fig. A.5. Path Evaluation Semantics

• **Node Operators.** As mentioned in the informal description of the language (see Section 3), extraction rules are not concerned by nodes themselves but

by values they carry. Node operators are extracting such values.

$$
\begin{array}{lcl}
\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, null & = & null \\
\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, [\,] & = & [\,] \\
\mathbb{E}_{\mathrm{N}}[\![\ \texttt{.txt}\ ]\!]\, node & = & \mathrm{getText}(node) \\
\mathbb{E}_{\mathrm{N}}[\![\ \texttt{.attr(}name\texttt{)}\ ]\!]\, node & = & \mathrm{getAttribute}(node,name) \\
\mathbb{E}_{\mathrm{N}}[\![\ \texttt{.src}\ ]\!]\, node & = & \mathrm{getHTMLSource}(node) \\
\mathbb{E}_{\mathrm{N}}[\![\ \texttt{.numberOf(}n\texttt{)}\ ]\!]\, node & = & \mathrm{count}\ [x \mid x \leftarrow attributes(node) \wedge name(x) = n] \\
\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, (h :: t) & = & (\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, h) :: (\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, t) \\
\mathbb{E}_{\mathrm{N}}[\![\ op,\ NSL_op\ ]\!]\, n & = & \mathbb{E}_{\mathrm{S}}[\![\ NSL_op\ ]\!]\, (\mathbb{E}_{\mathrm{N}}[\![\ op\ ]\!]\, n)
\end{array}
$$

Fig. A.6. Node Operators Semantics

We assume the existence of ancillary functions `getText`, `getAttribute` and `getHTMLSource`. These functions can be easily represented using the functions from the data-model. We do not describe them in details because they are not relevant to the semantics per se of the language. We could assume they are built-in. Informally, for `getText`, we start from the *node*, visit its children in a depth first strategy and concatenate the text values; for `getHTMLSource`, we do the same but also include attributes and tagging symbols.

• **NSL Operators.** NSL operators are functions that takes an NSL – and maybe other parameters – as an input and return an NSL. Examples of such operators are user-defined functions and built-in regular expression functions.

$$
\begin{array}{lcl}
\mathbb{E}_{\mathrm{S}}[\![\ op\ ]\!]\, null & = & op(null) \\
\mathbb{E}_{\mathrm{S}}[\![\ op\ ]\!]\, string & = & op(string) \\
\mathbb{E}_{\mathrm{S}}[\![\ op\ ]\!]\, (h :: t) & = & (\mathbb{E}_{\mathrm{S}}[\![\ op\ ]\!]\, h) :: (\mathbb{E}_{\mathrm{S}}[\![\ op\ ]\!]\, t) \\
\mathbb{E}_{\mathrm{S}}[\![\ op1,\ op2\ ]\!]\, n & = & \mathbb{E}_{\mathrm{S}}[\![\ op2\ ]\!]\, (\mathbb{E}_{\mathrm{S}}[\![\ op1\ ]\!]\, n)
\end{array}
$$

Fig. A.7. NSL Operators Semantics

Built-in regular expression operators in W4F are `regex` and `split`, as defined in Perl5 [28]. The evaluation of `regex` or `split` on *null* produces $null^{NSL}$. When applied to a string, `regex(pat)` will return *null* if the string does not match the pattern `pat`. If the string does match, `regex` will return the strings that correspond to the binders (if any) inside the pattern, or the string itself. When applied to a string, `split(pat)` will return the substrings that are separated by `pat` inside the string.

• **Conditions.** The denotational semantics of the extraction path language has carefully ignored conditions. It is not easy to plug them elegantly in this formal framework. One important thing to keep in mind is that conditions have no influence over the structure of the final result (in terms of nesting).

As far as the syntax is concerned, conditions are an extension of index ranges. When used in the `WHERE` clause of the extraction rule, variables appear alone, with no index range.

index range with condition	=	*index variable* `":"` *index range*
index variable	=	string

As far as the structure of conditions is concerned, there are two main constraints:

(i) it is always possible to sort conditions topologically and resolve them one at a time

(ii) if a non singleton range appears either on the left or right side of the index variable, it must be identical to the one present in the extraction path.

To make things simple, we will assume that conditions are being resolved first and that when the extraction paths are applied, the correct values for index variables is already known. Let us consider the evaluation of the index range $i : \alpha$. Conditions related to variable i (for paths in the `WHERE` clause that mention i or that mention variables that need to be resolved before i) are resolved to produce a list L_i of integers. This list is then transformed using[16] $\mathbb{R}(L_i, \alpha)$ to produce the NNL.

The path evaluation semantics can now be rewritten as:

$$\mathbb{E}[\![\ .\mathrm{tag}[i:\alpha]\ path\ ]\!]\ node \quad =$$

$$\mathbb{E}[\![\ path\ ]\!]\ \mathbb{R}([x \mid x \leftarrow children(node)\ \wedge name(x) = tag], \mathbb{R}(L_i, \mathbb{N}(\alpha)))$$

$$\mathbb{E}[\![\ \texttt{->}\mathrm{tag}[i:\alpha]\ path\ ]\!]\ node \quad =$$

$$\mathbb{E}[\![\ path\ ]\!]\ \mathbb{R}([x \mid x \leftarrow successor(node)\ \wedge name(x) = tag], \mathbb{R}(L_i, \mathbb{N}(\alpha)))$$

B Template-to-DTD translation semantics

The semantics of the translation from the template language into a DTD is defined by two functions:
$\mathbb{T}[\![\ \ ]\!]$ ($Template \rightarrow string \rightarrow DTD\,declaration$) translates a $Template$ into a DTD declaration (element or attribute).
$\mathbb{N}[\![\ \ ]\!]$ ($Template \rightarrow string$) simply returns the name of the top-level template.

For the record rule, we assume that a sequence of empty names corresponds to `EMPTY`. This might be the case when the templates describe attributes. For the list rule, the presence of multiple "`*`" in the left-hand side has no effect on the produced DTD.

[16] Even though the function $\mathbb{R}$ has been defined for NNL, we will extend it for lists of integers.

$$
\begin{array}{lcl}
\mathbb{T}[\![\ .tag(t_1\texttt{\#..\#}\ t_k)\ ]\!]\mathrm{n} & = & \texttt{<!ELEMENT tag (}\mathbb{N}[\![\ t_1\ ]\!],..,\mathbb{N}[\![\ t_k\ ]\!]\texttt{) >} \\
 & & \mathbb{T}[\![\ t_1\ ]\!]\mathrm{tag}\ ..\ \mathbb{T}[\![\ t_k\ ]\!]\mathrm{tag} \\
\mathbb{T}[\![\ .tag\texttt{*}\ t\ ]\!]\mathrm{n} & = & \texttt{<!ELEMENT tag (}\ \mathbb{N}[\![\ t\ ]\!]\ \texttt{)*>} \\
 & & \mathbb{T}[\![\ t\ ]\!]\mathrm{tag} \\
\mathbb{T}[\![\ .tag\ ]\!]\mathrm{n} & = & \texttt{<!ELEMENT tag (\#PCDATA)>} \\
\mathbb{T}[\![\ .tag\texttt{\^{}}\ ]\!]\mathrm{n} & = & \texttt{<!ATTLIST n tag \#CDATA \#IMPLIED>} \\
\mathbb{T}[\![\ .tag\texttt{!}att\ ]\!]\mathrm{n} & = & \texttt{<!ELEMENT tag EMPTY>} \\
 & & \texttt{<!ATTLIST tag att CDATA \#IMPLIED>}
\end{array}
$$

Fig. B.1. Template-to-DTD translation ($\mathbb{T}[\![\ \ ]\!]$)

$$
\begin{array}{lcl}
\mathbb{N}[\![\ .tag\ ]\!] & = & \mathrm{tag} \\
\mathbb{N}[\![\ .tag\texttt{\^{}}\ ]\!] & = & \emptyset \\
\mathbb{N}[\![\ .tag\texttt{!}att\ ]\!] & = & \mathrm{tag} \\
\mathbb{N}[\![\ .tag\texttt{*}t\ ]\!] & = & \mathrm{tag} \\
\mathbb{N}[\![\ .tag(t_1\texttt{\#..\#}\ t_k)\ ]\!] & = & \mathrm{tag}
\end{array}
$$

Fig. B.2. Template-to-DTD translation ($\mathbb{N}[\![\ \ ]\!]$

C NSL-to-XML translation semantics

We now describe how templates are applied to NSL to produce XML documents. The semantics is defined by one function $\mathbb{T}[\![\ \ ]\!]$: *template* $\rightarrow$*Nested String List* $\rightarrow$*bool* $\rightarrow$*string*. The function takes a template and an NSL to produce an XML document (string). The third argument is a boolean flag used to distinguish between NSL items that will produce attribute content ($flag = true$) and items that will produce element content ($flag = false$).

For each case, we describe with some pseudo-code how the XML document is produced.

Unlike the DTD mapping that will always produce a DTD, the XML mapping might fail if the template and the NSL do not match. There are two cases of mismatches: (1) when a record template does not get the correct number of elements and (2) when a leaf template gets a list instead of a string.

For the list rule, the presence of multiple "*" in the left-hand side will force the input nsl to be flattened (as many times as there are stars).

$$
\mathbb{T}[\![\ .tag(t_1\texttt{\#..\#}\ t_k)\ ]\!](nsl, b) \quad = \quad \begin{cases} \text{if size(nsl) != k then FAIL else} \\ \text{if } b = true \text{ then "" else} \\ \texttt{<tag}\ \mathbb{T}[\![\ t_1\ ]\!](nsl[1], true)\ ..\ \mathbb{T}[\![\ t_k\ ]\!](nsl[k], true)\ \texttt{>} \\ \mathbb{T}[\![\ t_1\ ]\!](nsl[1], false)\ ..\ \mathbb{T}[\![\ t_k\ ]\!](nsl[k], false) \\ \texttt{<tab/>} \end{cases}
$$

$$
\mathbb{T}[\![\ .tag\texttt{*}\ t\ ]\!](nsl, b) \quad = \quad \begin{cases} \text{if } b = true \text{ then "" else} \\ \texttt{<tag>} \\ \text{for i=1 to size(nsl)}\ \mathbb{T}[\![\ t\ ]\!](nsl[i], b) \\ \texttt{</tag>} \end{cases}
$$

$$
\mathbb{T}[\![\ .tag\ ]\!](nsl, b) \quad = \quad \begin{cases} \text{if } nsl \text{ instanceof } string \text{ then} \\ \text{if } b = true \text{ then "" else } \texttt{<tag> n </tag>} \\ \text{else FAIL} \end{cases}
$$

$$
\mathbb{T}[\![\ .attr\texttt{\^{}}\ ]\!](nsl, b) \quad = \quad \begin{cases} \text{if } nsl \text{ instanceof string then} \\ \text{if } b = true \text{ then } \texttt{attr="nsl"} \text{ else ""} \\ \text{else FAIL} \end{cases}
$$

$$
\mathbb{T}[\![\ .tag!att\ ]\!](nsl, b) \quad = \quad \begin{cases} \text{if } nsl \text{ instanceof } string \text{ then} \\ \text{if } b = true \text{ then "" else } \texttt{<tag att="nsl"/>} \text{ else FAIL} \end{cases}
$$

Fig. C.1. NSL-to-XML translation