web ecology: recycling html pages as xml documents using w4f

arnaud sahuguet
department of computer and information science
university of pennsylvania
sahuguet@saul.cis.upenn.edu

fabien azavant
école nationale supérieure des télécommunications
paris, france
fabien.azavant@enst.fr

abstract

in this paper we present the world-wide web wrapper factory (w4f), a java toolkit to generate wrappers for web data sources. some key features of w4f are an expressive language to extract information from html pages in a structured way, a mapping to export it as xml documents and some visual tools to assist the user during wrapper creation. moreover, the entire description of wrappers is fully declarative.

as an illustration, we demonstrate how to use w4f to create xml gateways, that serve transparently and on-the-fly html pages as xml documents with their dtds.

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 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 reviews, digests, links, etc.

as soon as we want to go beyond the basic mode of a browsing human, we need software applications and these need to access web data in a structured way. html was really designed to display information to a human user, so applications need either html wrappers [13] that can make the content of html pages directly available to them or a more structured (i.e. application-friendly) data format like xml [15].

the rise (yet to come) of xml will lead to some interesting ecological issues (html and xml can be seen as competing species in the web arena). proposals like xsl [16] already address the problem of translating xml into html. but a more interesting problem is to integrate legacy html pages in an xml environment, using html wrappers. some people have already argued that there is no more need for such wrappers because data sources will soon serve xml documents. in fact, there already are countless html pages on the web that need to be "recycled" as xml documents in order to take advantage of the xml "new world" that is being promised.

in this paper we argue that our toolkit can offer a simple and elegant solution to this ecological problem, and we show how to create xml gateways that serve transparently and on-the-fly - pages from html sources as xml documents with their dtd. we illustrate our approach using the cia world factbook site.

the rest of the paper is organized as follows. section 2 briefly presents the w4f toolkit. in section 3, we give an overview of our extraction language. section 4 describes our internal data model to store extracted data. the mapping to xml is explained in section 5. section 6 illustrates the visual support offered by the toolkit. finally, we summarize our work and compare it to other approaches.

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.

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.

the toolkit 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 the specification of each layer.

3 extracting information from an html page

in this section, we glimpse at some features of hel (html extraction language) used for the specification of the extraction layer. the full details can be found in [13]. features presented here after are motivated by the extraction of various pieces of information - such as country name, capital, government, etc - from a country entry in the cia world factbook, and are illustrated by the example of figure 1.

1 http://www.odci.gov/cia/publications/factbook
2 for more details about structure and content of country entries, you can visit http://www.odci.gov/cia/publications/factbook/fr.html
HEL is a DOM-centric [17] language where an HTML document is represented as a labeled graph. 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 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 node or a child of a node.

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, inside the <HTML> tag of the page. This type of navigation offers a unique/canonical way to reach each information token.

The second navigation is along the document flow, using the "->" operator. Path 'html.body.p[0]->pcdata[1]' will lead to the second chunk of text found in the depth-first traversal of the abstract tree starting from the first paragraph <P> in the body of the document. This operator is very useful to create navigation shortcuts.

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 collections 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]) or an index range (like [i:1], 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.

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. To capture this level of details, our language comes with standard regular expressions à la Perl [14] 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 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 1, match is used to extract separately from land area the value and the unit; split is used to extract a list of strings out of the industries data.

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 sub-paths at the same time. Forks can be applied in cascade. In the example of Figure 1, to extract both chief of state and head of government, we first follow path 'body.p[1]' and then fork into two sub-paths '->[x]->pcdata[1].txt' and '->[x]->pcdata[1].txt'. This is particularly useful when information spread across the page need to be put together.

Another key feature 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]) or an index range (like [i:1], 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.

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. To capture this level of details, our language comes with standard regular expressions à la Perl [14] 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 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 1, match is used to extract separately from land area the value and the unit; split is used to extract a list of strings out of the industries data.

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 sub-paths at the same time. Forks can be applied in cascade. In the example of Figure 1, to extract both chief of state and head of government, we first follow path 'body.p[1]' and then fork into two sub-paths '->[x]->pcdata[1].txt' and '->[x]->pcdata[1].txt'. This is particularly useful when information spread across the page need to be put together.

\[\text{Figure 1: extraction rules expressed using HEL, for the CIA World Factbook.}\]
4 Storing information using NSL

Within W4F, information is stored in Nested String Lists (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 from 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 rule of Figure 1, we can infer that the corresponding NSL will be a list of 5 items (4 top level forks). Items 0 and 1 will be strings. Item 2 will be a pair of strings (1 match with 2 bindings). Item 3 will be a pair of strings (2 match with 1 binding). Item 4 will be a list of strings (split).

As mentioned in the previous section, operators can be applied in cascade which can lead to highly complex structures.

5 Mapping information to XML

W4F comes with an automatic mapping to Java base types and their array extensions, but it works only for regular structures (list items must have the same structure). For more irregular NSLs, the user can define a Java mapping by providing some Java classes with valid constructors that can consume the NSL to produce Java class instances. But for the sake of this paper, we will focus on the XML mapping offered by the toolkit.

An XML mapping expresses how to create XML elements out of NSLs. It is important to keep in mind that the structure of XML elements we can generate is constrained by the structure of the NSL itself. Such a mapping can be seen as a set of rules that determine how to consume NSLs to produce XML elements.

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
```

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 an attribute of a bachelor element. We show below how these cases can be specified in the language and the output XML element for an input NSL string (say "France"):

- `cy` → `<CY>France</CY>`
- `cy .ID` → `<CY ID="France" _ _ _>`
- `cy!ID` → `<CY ID="France"/>

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

A record template like `<Country(t1 ... t_n)>` consumes a list of n NSL items. It first creates a new element `<Country>` and applies each inner template t_i to its corresponding i-th list item, using concatenation. Finally the element is closed with `</Country>`.

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 to consume the NSL and produce a well-formed XML document. The construction of the DTD is straightforward.

![XML_MAPPING](http://www.odci.gov/cia/publications/factbook/fr.html)

Figure 2: a possible XML mapping for our Web source.

As an illustration, we present in Figure 2 a possible XML mapping for the CIA Web source. The output XML document is shown in Figure 3.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE W4F_DOC [...
```

Figure 3: the "recycled" document with its DTD for http://www.odci.gov/cia/publications/factbook/fr.html.
6 Support via visual interfaces

In order to make the specification of our wrappers fast and easy, we provide some visual tools – wizards – that assist the user during the various stages of the wrapper construction.

A critical part of the design of the wrapper is the definition of extraction rules since it requires a good knowledge of the underlying HTML. The role of the extraction wizard (see Figure 4) is to help the user write such rules. For a given HTML document, the wizard feeds it into W4F and returns the document to the user with some invisible annotations (the document appears exactly as the original).

Figure 4: the extraction wizard used on two different pieces.

On Figure 4, when the user points to "Industries", the corresponding text element gets highlighted (the user can identify information boundaries enforced by the HTML tagging) and the canonical extraction rule pops-up. 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 (Figures 4 and 1).

Another useful interface permits to test and refine the wrapper interactively before deployment. Figure 5 shows the wizard (a Java applet here) which visualizes the 3-layer architecture of the wrapper.

Figure 5: a visual view of a wrapper with its 3 layers.

As of this writing, an XML-mapping wizard is under development.

7 Conclusion and related 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, index 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 that offers the generation of ready-to-use Java classes and some visual tools to assist the user.

Compared to other approaches [10, 11], we do not use a grammar-based approach for extraction but rely on the 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 [8, 3]. Unlike We b-OQL [4], we do not try to query the Web but simply extract structure from Web information sources: querying is the concern of the upper-level application.

Our tackling of XML is different from the one of XML-QL [7] (based on patterns and explicit constructs) because we derive it from our extraction process that handles HTML pages that have no explicit structure. As a consequence we cannot for instance express object sharing (using TIDREF). This is a tradeoff to pay for: we think that this is not the role of the wrapper but of the upper-level application.

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 TIS-MIS [9], Kleidi [6], Garlic [12], etc.

Our goal with W4F is to offer extraction capabilities powerful enough to handle any kind of HTML structure and thus make the construction of wrappers fast and elegant, as illustrated by our XML gateway example for the CIA Web World Factbook.

The toolkit is freely available at http://db.cis.upenn.edu/W4F. On-line examples of W4F applications (including the wrapper presented here) can be found at the same location.

References


