Wrapping Web Pages into XML Documents

A Practical Experience and Comparison of Two Tools

CMIS Technical Report 01/199

Sabine Jabbour[1]
sabine@jabbour.net
Anne-Marie Vercoustre
Anne-Marie.Vercoustre@csiro.au

February 2001

TED Group, CMIS

Abstract

The notion of wrapping a web server into XML documents is driven from the need for structured data that can be used by a variety of applications. The web contains vast amounts of information that is useless to most applications since it is mainly targeting a human audience. A solution to this would be to automate the browsing process and then convert the extracted information into a more suitable format – like XML. This is called wrapping. We have used two different tools to wrap several tourist sites into XML The tool we have been using are Norfolk, a system developed since 1997 by the CSIRO TED group and W4F, initially developed at the University of Pennsylvania, now a commercial product. This report describes our practical experience with the tools and makes comparison between them.

1. Introduction

XML (Extensible Markup Language) allows data to be stored within a document and information about the data to accompany it. In comparison with HTML (HyperText

[1] This work was developped when Sabine Jabbour, student at Monash Univerity, was doing her internship at CSIRO.

Markup Language), which is mainly specifying the layout of a page, XML separates the data from the layout. XML documents can be viewed in much the same as any HTML page by defining a stylesheet for the page. However, XML tags are meaningful with respect to their content and these tags can be defined by the application that requires them.

The word 'wrapper' originates from the database community. A wrapper in this context is used as a mediator between several databases and an application. In a similar way, in the web environment, a wrapper converts information from HTML documents into information stored as a data-structure (like XML).

Although there are promising researches on wrapper induction ([1][3]) to automatically generate wrappers from a few examples, the technique is not yet quite effective to write real wrappers for Web sites. The approach we chose was to write a program using a scripting language designed for writing wrappers. There are two tools we used for this purpose – Norfolk and W4F. Norfolk is a language and a system developed by the TED group in 1997 initially for creating virtual documents from heterogenous sources [5]. It has recently been extended to cater for the creation of XML documents for the purpose of wrapping. W4F (World Wide Web Wrapper Factory) began as a research tool developed by the University of Pennsylvania (Penn Database Group) [3]. It has now become a wrapping product.

We have used Norfolk and W4F to wrap different tourist sites and get XML data for another project demonstrator. The sites we wrapped were mostly www.fodors.com and www.intown.com.au. Automatic wrapping is worthwhile only for sites that offer a lot of data in a regular format (e.g. a list of hotels) or a large set of similar pages (e.g. hotels for different cities). It is generally the case when a site is generated from a database. However the underlying HTML page may not be as regular as it looks like from a quick look at the page displayed by the browser.

In this paper, some of the difficulties encountered with Norfolk and W4F will be outlined. In addition, they will be compared as wrapper tools in general. This has involved some testing for a variety of wrappers written for the same web server using each of the two tools.

2. The General Tasks of a Wrapper in the Web Environment

A wrapper has three main tasks: first, to *retrieve* a web document, to *extract* the relevant information from the web page; third, and to *map* the information into the XML format required by any specific application.

2.2 A simple example

Let take the example of the tourist Web site http://www.fodors.com/ and the page that lists hotel information for Melbourne. The actual page[2] is shown in Figure.1. The wrapper must first retrieve the page using its URL. Then it has to extract the relevant

[2] The screen dumps has been taken in 2001 and the site has changed since.

information in the page. The page display details for 19 hotels in Melbourne and for each of them we want to get its name, address, phone, fax, price range and description.

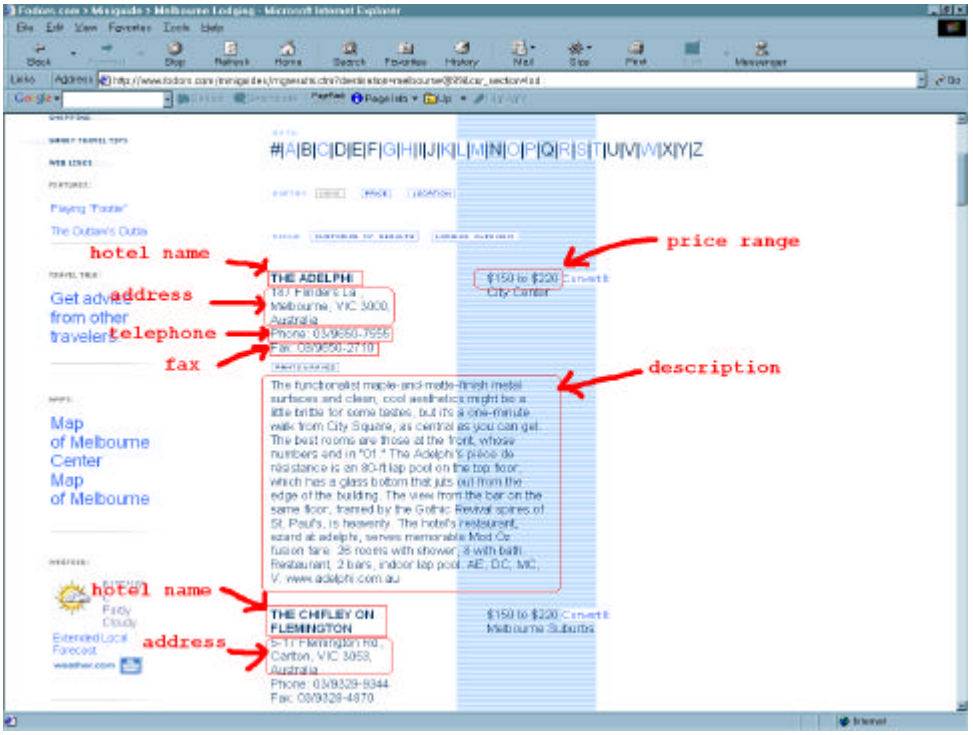

Figure.1 Example of page to be wrapped

The objective of the wrapper is to extract the relevant information from the HTML page and to transform it into a format that can be easily reused by application, in our case XML. Figure 2. shows on the left side the HTML source of the page to be wrapped and on the right side the result in XML, after the information has been extracted.

The XML document could mention the DTD the document has to conform as well as a style sheet to display the XML document with a browser.

The same wrapper can be used to wrap hotel information available for other cities on the site, by just changing the URL of the page that is passed to the wrapper as a parameter.

2.3 Tree based wrappers

The two wrappers we have used worked from the tree representation of the Web pages, therefore taking advantage of the hierarchical structure of the tags when present. Both Norfolk and W4F have developed a language that uses the notion of *tree path* to locate the information to be extracted. Figure 3. shows a simplified version of the tree associated with the HTML page presented in figure.2.

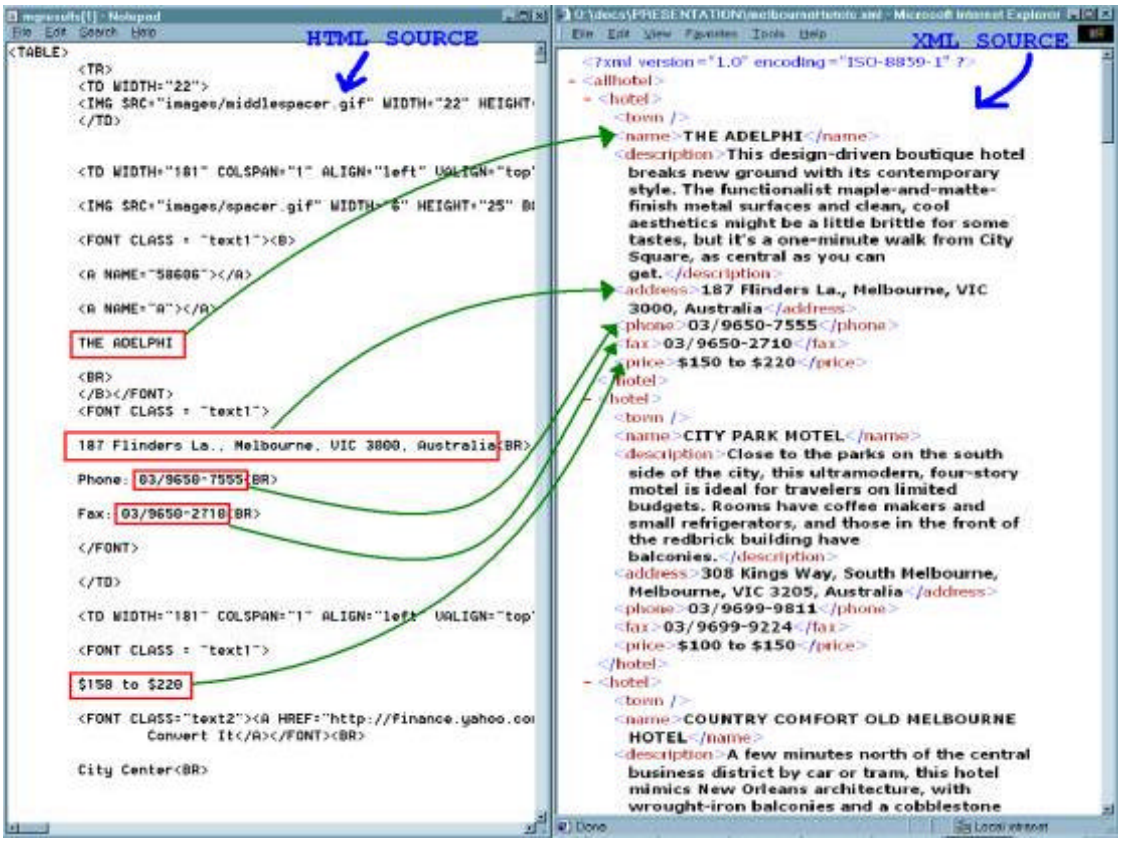

Figure.2 Wrapping HTML text into XML data

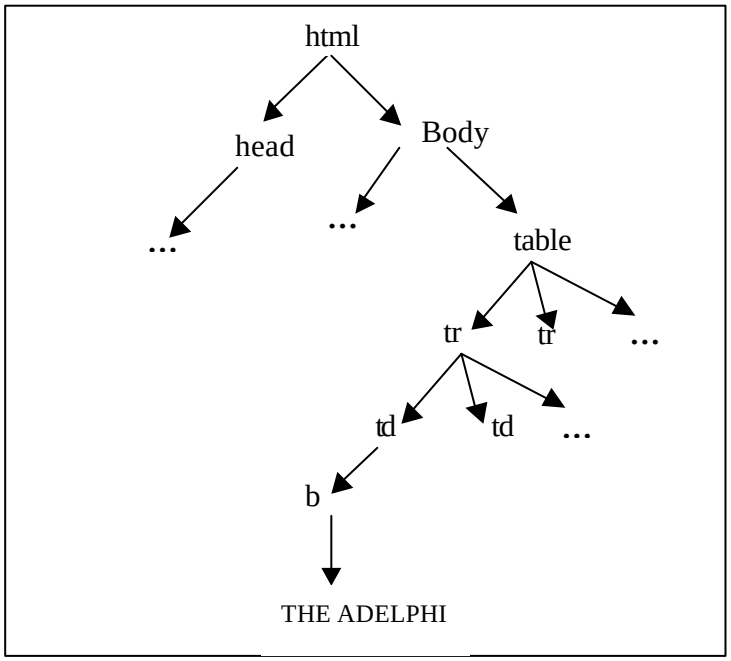

Fig.3 Tree representation of an HTML page

An *absolute path* in a tree is an expression that describes how to access a specific element by traversing the tree from child to child. For example, if we suppose there is only a table in the document, the path:

html.body.table.tr[0].td[0].b.text (1)

will access the node that contains the string “THE ADELPHI”.

A *relative path* is a path that is described using the descendants rather than the direct childs. For example, in Norfolk:

html..table..td

will retrieve all the elements (sub-trees) with label *td* in all the *table* elements.

In Norfolk and W4F paths can be selected when some conditions are fulfilled (expressed with a *where* condition. For example, using Norfolk syntax:

select html..table as $table where $table..td contains “THE ADELPHI” (2)

will also select the table that contains the string "THE ADELPHI" .

The XML Path Language (XPath) [6] is a W3C recommendation for addressing parts of an XML document using *location paths*. Xpath are more complex than the one described here since they use more than the two *axes* "child" and "descendants" for traversing the tree. Norfolk and W4F have been designed before XPath and can be seen as a simplified version of XPath using a different syntax.

3. Evaluation of Norfolk vs. W4F

Extending the evaluation criteria proposed in [3] for evaluating Wrapper induction, we will evaluate Norfolk and W4F (Wrapper scripting), according to:

- The expressiveness of the language at the micro level, i.e. how the scripting language is well designed for wrapping single HTML pages
- The general architecture and expressiveness of the language at the macro level, i.e. how the language is well suitable to wrap a set of linked pages and map extracted information into a given XML format.
- The efficiency of the actual wrappers, i.e the time it takes to wrap a (set) of page(s), and how robust is the wrapper against changes in the page.

3.1 Expressiveness

The expressiveness of a wrapper scripting language has to be evaluated for the three tasks of accessing, extracting and mapping information. First Norfolk allows the integration of the three tasks within the wrapper definition, whereas W4F has a clear separation between them. We will see respective advantages of each approach while comparing the two wrappers in more details along the three tasks mentioned above.

3.1.1 Retrieving a web document:

Both Norfolk and W4F allow the retrieval of several pages within the one wrapper. However, W4F only allows extraction from one of the retrieved pages within the one wrapper, while Norfolk allows extraction from any number of retrieved pages within the same wrapper.

3.1.2 Extracting information from web documents

Extraction in both Norfolk and W4F is done using the paths of the relevant information in the parsed HTML tree. In addition, some basic string matching rules are used (regular expressions).

This process begins by pruning the tree to reach the relevant subtree and then applying the matching rules on the parts of the tree that contain the necessary information.

In the following example, the wrapper extracts the text of a specific row (*tr*) of a *table* where the second *tr* element contains the string "We found" and some other *tr* element(s) contain the string "Convert it".

W4F example:

```
info = html.body->table[i].tr[j:0-]->b.txt
    WHERE html.body->table[i].tr[1].txt =~ "We found"
    AND html.body->table[i].tr[j].txt =~ "Convert It";
```

Norfolk example:

```
<?define $info as select $tr.#VALUE
        from $page.body..table as $table, $table.tr as $tr,
        where $table.tr[1] contains  "We found" and
            $tr contains "Convert It">
```

There is no real difference, other than syntactic, between how the two tools find the proper table and extract the specific information. The major difference is that Norfolk introduces explicit variables ($table and $tr) to make the joints on *table* and *tr*, while W4F use index variables (i and j).

We are going to compare a bit further the expressiveness of the two languages for path expressions.

Navigating the document hierarchy

W4F and Norfolk have relatively similar ways of navigating the tree representation of the parsed web page.

For example, *html.body.table[1].tr[0].td[2]* would be interpreted in the same way by both W4F and Norfolk, although the default for the index value is different.

e.g. html.body.table[1].tr[0].td would be interpreted as:

W4F: *html.body.table[1].tr[0].td[0]* -- *W4F returns the first td element*

Norfolk: *html.body.table[1].tr[0].td[*]* – *Norfolk returns all td elements*

W4F has three ways of accessing nodes in a tree:

"." indicates a direct child

"->" will search any part of the tree using depth-first traversal

"-/> will only search within the scope of the current sub-tree

Norfolk has only two ways:

"." indicates a direct child

".." will search within the scope of the current sub-tree

W4F has an extraction wizard, the purpose of which is to give the absolute tree path for any part of the web page. The idea behind this wizard is very good because, when writing a wrapper, you need first to locate the information you are interested in retrieving, and the HTML structure might be very complex. However, this tool only allows you to see these paths if the mouse is scrolled over the relevant item (the path disappears too quickly) or by viewing the source of the page after it has been run through the extraction wizard. This would be a very powerful tool if integrated into a wrapping user interface. Norfolk has no such wizard.

In Norfolk, extraction rules return nested lists of trees on which one can recursively apply other extraction rules. Appropriate functions (namely text, sgml, xml) will return the text or the underlying tagged source associated with the node. The textual value of a node or an attribute can also be extracted directly with an expression such as

html.body.table[1]..td[2].#VALUE

which is equivalent to:

text(html.body.table[1]..td[2])

W4F is less interested in the nodes themselves, but more interested in the information they carry. This means it does not let you extract sub-trees for doing further processing. Any W4F instruction is a rule for extracting some text, while Norfolk offers a full language for tree extraction and transformation.

Semantic difference in basic path expressions

The fundamental difference between the two languages is that path expressions in Norfolk *never* fail – they only return an empty list if the specified expression does not

correspond to any actual node. With W4F, the expressions must refer to a real path in the tree for the wrapper to execute successfully. To avoid W4F to fail on irregular or incomplete data, defaults (exceptions) must be specified for *all* extraction rules. This is because the majority of websites, even ones that are relatively consistent, will have some fields of information missing.

An example of this in W4F is:

html.body->table[0].tr[2]->font[1].txt, match("(.?) Fax"), default("");*

In case a restaurant does not give a fax number, the wrapper will generate an empty string. If the default rule is not specified, W4F will fail to wrap this page.

HTML and XML documents are generally semi-structured documents, e.g. documents that present an irregular and incomplete structure. We believe that Norfolk semantic for interpreting the path expressions is better suited to semi-structured documents than W4F semantic. Norfolk semantic is similar to the one defined for XPATH expressions as defined by W3C.

3.1.3 Mappings

W4F requires an explicit definition of the mapping rules which must correspond in order and number to the extraction rules. An additional feature is that W4F generates a DTD from these mapping rules. The advantage is that it guaranties that the XML result will always conform with the DTD. This is simple and useful, but restrictive for most applications.

Indeed it is often necessary to have optional elements in a DTD and this is not possible with W4F since the mapping rules are relatively simplistic. It is also a common problem that an application has a DTD which the XML documents generated should conform to.

Norfolk does not provide any DTD as a result of wrapping. Instead we explicitly map to a previously defined DTD format. This was much appropriate when you want to apply one DTD to all the kinds of tourism information we wrapped. How ever Norfolk does not verify that the XML document resulting from the wrapping conforms to the DTD. This has to be checked separately until your wrapper is validated.

Another situation that is often encountered when wrapping a web server is that you want to add in static values for some elements of the XML document. This is because you do not need or want to extract them from the pages. For example, when wrapping a hotel website (see www.fodors.com), each page may contain a list of hotels in a particular city. The name of each city does not need to be extracted since it is already known (or found in the parameter for the page). In this case, W4F will not allow the addition of static information in the mapping. This is a great disadvantage and limits W4F immensely in comparison with Norfolk's flexibility in this respect. Norfolk being a tool for creating virtual documents, permits any inclusion of information.

Example in W4F:

html.body->table[1].tr[3]->web.txt, default("http://www.fodors.com")

Note here that the only way of creating a static value is cheating W4F with a default value. This allows you to create a tag with a static value that is to be repeated for all the elements this extraction rule maps into, which might not be quite desirable. The example below shows that Norfolk only needs the wrapper to have a tag created with the value in it, and it will be embedded into the document generated.

Example in Norfolk:

<source>http://www.fodors.com</source>

In resume, Norfolk and W4F have an equivalent expressiveness regarding the path expressions, W4F being a bit more declarative and simpler, but Norfolk language offering a better semantic for extraction from semi-structure documents, as well as being more flexible for mapping into predetermined DTD formats.

3.2 General architecture and Web environment

Writing wrappers often involve extracting data from a set of pages rather than a single page. You may for example wrap pages that describe a list of movies, or hotels in various cities from a given server. Wrappers based on scripting language or on induction techniques can only work for wrapping pages that are very similar in their structure.

As describe before, good wrappers should be flexible enough to wrap a large amount of pages that present a similar but necessarily identical content structure. This is what we called micro-level wrapping that deals with wrapping a single type of page.

However very often want to combine information from different pages which amount in being able to follow links from one page to another. A very common situation is to wrap an initial page that contains a list linked to other pages that we want to wrap. A example is a list of cities that link to each city page which in turn contains links to a page for hotels and a page for restaurants.

Wrapping the site requires to be able to combine several wrappers which each will wrap different type of pages. In some way you need to combine crawling facility with wrapping techniques.

Norfolk is well designed for wrapping a set of pages into XML data and calling recursively several wrappers. The set of pages may be linked or not. Although an extension to Norfolk language to support declarative traversal of links has been proposed in [8], it has never been implemented. The current implementation of link traversal is very procedural and require to explicitly retrieve the link URL before requiring the proper http connection.

W4F offers a very limited way of following links and requires a Java program to combine specialised wrappers. It is not possible either with W4F language to build a single wrapper that extracts information from several pages.

In conclusion, Norfolk is better suited to create large application wrappers able to extract information from a variety of pages and to combine them.

3.3 Testing performance

Since a wrapper is usually used dynamically and regularly to get updated data, it is important that it is efficiant in doing it. We did some testing for times for both Norfolk wrappers (against each other) and two wrappers written using Norfolk and W4F (for the same web server).

3.3.1 Norfolk Tests

Three different Norfolk wrappers were tested against each other. These were wrapping information from http://www.fodors.com/ for 25 cities and get the list of restaurants, hotels and activities for each city. Norfolk generated 25 XML documents for each wrapper. The tests were run 11 times and on different times of the day. The table on the next page shows the details of the tests. The wrappers themselves can be found in Appendix A.

The only odd result was obtained in the test run on 11/01/01 at approximately 10:30am. This was an unexpected result and is most likely due to a problem with the server at the time. The averages calculated did not include this result.

The most notable difference is that the restaurants wrapper took the longest and the activities wrapper took the least time in wrapping. This can be attributed to the amount of extraction and mapping the wrapper has to do. First, the restaurant wrapper had to extract the most information, the activities one extracted the least. The wrapper retrieves the page *once* only, so that is not an affecting factor in this case.

Time of wrapping	Accommodation wrapper	Restaurant wrapper	Activities wrapper
10/01/01 ~11:00am	5’13’’	6’17’’	2’04’’
10/01/01 ~2:45pm	4’48’’	6’05’’	2’01’’
10/01/01 ~5:45pm	4’22’’	5’36’’	1’49’’
11/01/01 ~10:30am	10’07’’	19’48’’	3’41’’
11/01/01 ~2:30pm	4’43’’	6’23’’	2’08’’
11/01/01 ~6.15pm	4’21’’	5’30’’	1’51’’
12/01/01 ~12.30pm	4’41’’	5’53’’	1’56’’
12/01/01 ~3.30pm	4’42’’	5’55’’	2’10’’
AVERAGE	~ 4’41’’	5’57’’	~ 1’59’’

Table 1. Execution time for different Norfolk wrappers

These tests show that the time of the wrapping itself (extraction and mapping) is significant against the total time that includes the server access and the download of the pages. It was important to set this before carrying tests for comparing Norfolk and W4F.

3.3.2 Norfolk vs. W4F runtime tests

This phase of testing was for the purpose of determining whether Norfolk was slower than W4F. This was the expected result, since Norfolk language is interpreted whilst W4F is compiled into Java classes. However, as table 2. indicates, there is no considerable difference between how the two performed on average. In fact, Norfolk is slightly, yet not significantly, faster.

The two wrappers were both using the same tourism website (http://www.fodors.com). However, only the hotel pages were tested, for the same twenty-five cities.

Time of wrapping	W4F	NORFOLK
19/01/01 ~11:30am	4’51’’	5’29’’
19/01/01 ~1:30pm	5’00’’	4’58’’
19/01/01 ~3:00pm	4’54’’	5’08’’
19/01/01 ~4:15pm	4’31’’	4’58’’
23/01/01 ~5:15pm	5’20’’	5’07’’
24/01/01 ~10:15am	4’44’’	5’12’’
24/01/01 ~1:10pm	4’59’’	5’04’’
24/01/01 ~3.15pm	6’22’’	4’55’’
25/01/01 ~10:45am	8’07’’	6’04’’
25/01/01 ~12:15pm	4’37’’	4’54’’
25/01/01 ~3:15pm	5’45’’	6’08’’
AVERAGE	~ 5’21’’	~ 5’16’’

Table 2. Runtime comparison between Norfolk and W4F

4. Robustness towards changes

General studies suggest that web pages, on average, are prone to changes every month. The rate of change is particularly high for commercial sites that use generation technique (from database) for page generation since it simplifies the maintenance and changes can be made consistently at a low cost.
Wrappers, such as Norfolk and W4F, that strongly rely on the underlying HTML tag structure to locate information are very vulnerable to changes and can be broken very easily.

It is a well-known problem that references within trees are hard to maintain when the tree is modified. Much attention to the problem has been paid in the Hypertext and Web community, specifically for trying to maintain semantic links within documents. In [4] a solution is proposed, using a node signature based on a combination of the absolute path of the node, the string value of the node and the one of its neighbours, as well as an associated repair algorithm.

The problem in wrappers is not exactly the same as for referential links. Indeed, for referential links one tries to keep the reference on an actual content in order to keep the link semantically relevant. If the content of the target node has changed too much, the link may become semantically invalid. In wrappers, on the opposite, one wants to maintain location references on *placeholders* for content, which is expected to have changed (e.g. the price of a product). The value of the content cannot be part of the signature of the node, although the content of a neighbour node can be used.

Incomplete path expression with condition like those presented in section 3.2.1 are more robust to structural change than absolute paths from the root. However they are not strong enough to resist big changes and even small changes can break a wrapper. When this happens the wrapper should at least be able to locate and report the problem.

We have started to investigate signatures that compare value (string) extracted by the wrapper with a domain vocabulary and will complain when a wrapped value is not of the expected type. More research is needed to make wrappers much more robust to change than they are today.

5. Conclusions and Future Work

Wrapping, in general is something that is aimed at making the job of gathering and converting information more efficient. For this, a wrapping tool must be easy to use for the human who must write the wrapper. It must also be fast in wrapping.

Our conclusion is that W4F and Norfolk are very effective in wrapping Web sites and relatively easy to use. The two tools offer a comparable language for wrapping single Web pages (comparable expressiveness), and take about the same time to wrap one page (comparable efficiency). Norfolk supports more complex wrapping from sets of linked pages and can run in server or batch mode. W4F offers a wizard interface that is helpful for beginners.

Although Norfolk was found to be more flexible and powerful to write actual wrappers, it is lacking a user-friendly interface similar to W4F Wizard, but possibly better integrated with the actual tool.

Both tools need to develop robust techniques to deal with frequent changes in the wrapped sites if the sites have to be wrapped regularly.

6. Bibliography

[1] Brad Adelberg, Mattiew Denny, Nodose version 2.0, SIGMOD'99

[3] Fabien Azavant, Arnaud Sahuguet, "World Web Wrapper Factory (W4F), User Manual", April 2000.

[3] Nicholas Kushmerick, "Wrapper induction: Efficiency and expressiveness", in *Artificial Intelligence* 118, (2000) 15-68, Elsevir (Ed.)

[4] Thomas A. Phelps and Robert Wilensky, “Robust Intra-document locations”, in *Proc. of WWW9 Conference*, Amsterdam, 1999.

[5] Anne-Marie Vercoustre and François Paradis, A Descriptive Language for Information Object Reuse through Virtual Documents, in *4th International Conference on Object-Oriented Information Systems (OOIS'97)*, Brisbane, Australia, pp299-311, 10-12 November, 1997.

[6] J. Clark and S. DeRose (eds), *XML Path Language (Xpath) Version 1.1*, W3C Recommendation, Novembre 1999, http://www.w3c.org/TR/xpath.

[7] Areneus: A tool for getting XML from HTML. http://www.dia.uniroma3.it/Araneus/

[8] Anne-Marie Vercoustre and François Paradis, *Reuse of Linked Documents through Virtual Document Prescriptions*, in Lecture Notes in Computer Science 1375, Proceedings of Electronic Publishing '98, Saint-Malo, France, pp499-512, 1-3 April, 1998.

6. Appendix A

Norfolk hotel wrapper

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="hotels.xsl"?>
<!DOCTYPE allhotel SYSTEM "tourism.dtd">

<?param "debug" as $debug default "no">
<?param "town" as $town default "melbourne@99">
<?define $page as url(str("http://www.fodors.com/miniguides/mgresults.cfm?",
                    "section_list=ove,sig,din,lod,nig,sho,tra,web,fea&destination=",
                     $town, "&cur_section=lod"))>
<?define $table as select $h from $page.body..table as $h
              where $h.tr contains "We found">
<?define $hotels as $table.[#FIRST]>
<allhotel>
<?map $i[$j] in $hotels..tr>
<?if $i contains "Convert">
<?begin>
<hotel>
<?define $text as $i.td[1]>
<?define $text1 as $i.td[2] exclude ..a>
<?– GETTING SOME HOTEL INFO AND SPLITTING IT  -->
<?– Getting town name -->
<?define $town2 as $town matches "(.*)\@">
<town>
          <?$town2.#ANY.#ANY[#LAST]>
</town>
<?define $text2 as strsplit(sgml($text exclude #ROOT..a), "</FONT></TD>")>
<?– Getting hotel names -->
<?define $text3 as $text2 matches "<b>(.*?)<BR></b>">
<name>
     <?$text3.#ANY.#ANY[#LAST]>
</name>
<?– Getting Description when NOT debugging -->
<?if $debug eq "no">
<?begin>
<description>
          <? str($hotels..tr[$j+2].td[1])>
</description>
<?end>
<?– Debugging Description -->
<?if $debug eq "yes">
<?begin>
<description>
          <?define $check1 as str($hotels..tr[$j+2].td[1])>
          <?define $check2 as str($hotels..tr[$j+1].td[1])>
          <?define $check3 as str($hotels..tr[$j+3].td[1])>
          <?if $check1 contains "Hotel|Hotels|hotel|hotels|Room|Rooms|room|
                    rooms|Restaurant|Restaurants|restaurant|restaurants">
                    <?begin>
          The correct description was found.
                    <?end>
          <?else><?if $check2 contains "Hotel|Hotels|hotel|hotels|Room|Rooms|room|
```

```
                rooms|Restaurant|Restaurants|restaurant|restaurants">
                <?begin>
        WARNING: The description was found in the previous tr.
                <?end>
        <?else><?if $check3 contains "Hotel|Hotels|hotel|hotels|Room|Rooms|room|
                rooms|Restaurant|Restaurants|restaurant|restaurants">
                <?begin>
        WARNING: The description was found in the next tr.
                <?end>
        <?else>WARNING: The extraction path for the description is invalid.
</description>
<?end>
<?– Getting address -->
<?define $text3 as $text2 matches "</b>[<|/|A-Z|a-z|0-9|\"|=|>| ]*(.*?)<BR>">
<address>
        <?$text3.#ANY.#ANY[#LAST]>
</address>
<?– Getting Phone number -->
<?define $text3 as $text2 matches "Phone:(.*?)<BR>">
<phone>
        <?$text3.#ANY.#ANY[#LAST]>
</phone>
<?– Getting Fax number -->
<?define $text3 as $text2 matches "Fax:(.*)<BR>">
<fax>
        <?$text3.#ANY.#ANY[#LAST]>
</fax>
<?– Getting Price Range -->
<?define $text2 as strsplit(sgml($text1), "<FONT")>
<?define $text3 as $text2.[1] matches "[<|/|A-Z|a-z|0-9|\"|=|>| ]*(.*)">
<price>
        <?$text3.#ANY.#ANY[#LAST]>
</price>
<source>
        http://www.fodors.com
</source>
</hotel>
<?end>
</allhotel>
```

Norfolk restaurant wrapper

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="restaurants.xsl"?>
<!DOCTYPE allrestaurants SYSTEM "tourism.dtd">

<?param "town" as $town default "lisbon@89">
<?define $page as url(str("http://www.fodors.com/miniguides/mgresults.cfm?",
                  "section_list=ove,sig,din,lod,nig,sho,tra,web,fea&destination=",
                   $town, "&cur_section=din"))>
<?define $table as select $h from $page.body..table as $h
           where $h.tr contains "We found">
<?define $restaurants as $table.[#FIRST]>
<allrestaurants>
<?map $i[$j] in $restaurants..tr>
<?if $i contains "Convert">
<?begin>
<restaurant>
<?define $text as $i.td[1]>
<?define $text1 as $i.td[2] exclude ..a>
<?– GETTING SOME RESTAURANT INFO AND SPLITTING IT  -->

<?– Getting town name -->
<?define $town2 as $town matches "(.*)\@">
<town>
                                        <?$town2.#ANY.#ANY[#LAST]>
</town>
<?define $text2 as strsplit(sgml($text exclude #ROOT..a), "</FONT></TD>")>
<?– Getting restaurant names -->
<?define $text3 as $text2 matches "<b>(.*?)</b>">
<?define $temp as $text3.#ANY.#ANY[#LAST]>
<name>
                              <?define $temp1 as $temp strreplace "<[^>]*>" with "">
                                        <?str($temp1)>
</name>
<?– Getting Description -->
<description>
                                        <?str($restaurants..tr[$j+2].td[1])>
</description>
<?– Getting address -->
<?define $text3 as $text2 matches "</b>[<|/|A-Z|a-z|0-9|\"|=|>| ]*(.*?)<BR>">
<address>
                                         <?$text3.#ANY.#ANY[#LAST]>
</address>
<?– Getting Phone number -->
<?define $text3 as $text2 matches "Phone:(.*?)<BR>">
<phone>
                                        <?$text3.#ANY.#ANY[#LAST]>
</phone>
<?– Getting Fax number -->
<?define $text3 as $text2 matches "Fax:(.*)<BR>">
<fax>
                                        <?$text3.#ANY.#ANY[#LAST]>
</fax>
<?– Getting Price Range -->
<?define $text2 as strsplit(sgml($text1), "<FONT")>
```

```
<?define $text3 as $text2.[1] matches "[<|/|A-Z|a-z|0-9|\"|=|>| ]*(.*)">
<price>
                                        <?$text3.#ANY.#ANY[#LAST]>
</price>
<source>
                                            http://www.fodors.com
</source>
</restaurant>
<?end>
</allrestaurants>
```

Norfolk activities wrapper

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="activities.xsl"?>
<!DOCTYPE allactivities SYSTEM "tourism.dtd">

<?– GETTING ACTIVITIES PAGE FROM fodors.com -->
<?param "town" as $town default "melbourne@99">
<?define $page as url(str("http://www.fodors.com/miniguides/mgresults.cfm?",
                  "section_list=ove,sig,din,lod,nig,sho,tra,web,fea&destination=",
                   $town, "&cur_section=sig"))>
<?– GETTING TABLE WHERE IMAGE "top sights and activities" CAN BE FOUND -->
<?define $table as select $h from $page.body..table as $h
                                    where $h.tr..img.src contains "sigheader.gif">
<?define $activities as $table.[#FIRST]>
<allactivities>
<?– LOOPING WITHIN THE ABOVE TABLE TO GET ALL THE ACTIVITIES AND THEIR
DETAILS -->
<?map $i[$j] in $activities..p>
<?begin>
<?define $text as sgml($i)>
<?if $text contains "<p><b>">
<?begin>
<item>
<?– TYPE OF INFORMATION BEING EXTRACTED -->
<type>
    Activity
</type>
<?– GETTING TOWN NAME -->
<?define $town2 as $town matches "(.*)\@">
<town>
      <?$town2.#ANY.#ANY[#LAST]>
</town>
<?– DEFINING DESCRIPTION PARAGRAPH FOR EXTRACTING SOME INFO -->
<?define $temp as $text matches "</b>(.*)</p>">
<?define $desc as $temp.#ANY.#ANY[#FIRST]>
<?– GETTING ACTIVITY NAME -->
<?define $name2 as $text matches "[<p>|<b>]<b>(.*?)[.|</b>|<b>]</b>">
<name>
                                <?define $name1 as $name2.#ANY.#ANY[#LAST]>
                        <?define $name as $name1 strreplace "<b>|</b>|</b|<i>|</i|</i>" with "">
                                                <?str($name)>
</name>
<?– GETTING ACTIVITY PHONE NUMBER -->
<?define $phone as ($desc matches "tel[.](.*?)[.|,]")>
<?define $phone1 as ($desc matches "PHONE: </FONT>(.*?)[.|,| |;|)]")>
<phone>
                        <?$phone.#ANY.#ANY[#LAST]><?$phone1.#ANY.#ANY[#LAST]>
</phone>
<?– GETTING ACTIVITY DESCRIPTION -->
<?define $desc2 as $desc matches "[.|</b>]*(.*)</p>">
<description>
                                <?define $temp as $desc2.#ANY.#ANY[#LAST]>
                              <?define $temp1 as $temp strreplace "<[^>]*>" with "">
                            <?define $temp2 as $temp1 strreplace "&amp;#" with "&#">
                                                <?str($temp2)>
```

</description>
<?– INFORMATION SOURCE -->
<source>
http://www.fodors.com
</source>
</item>
<?end>
<?end>
</allactivities>