A Supervised Visual Wrapper Generator for Web-Data Extraction

Xiaofeng Meng, Haiyan Wang, Dongdong Hu
School of Information
Renmin University of China,
Beijing 100872, China
xfmeng@ruc.edu.cn

Chen Li
School of Information and CS
University of California, Irvine,
CA 92697-3425,USA
chenli@ics.uci.edu

Abstract

Extracting data from Web pages using wrappers is a fundamental problem arising in a large variety of applications of vast practical interest. In this paper, we propose a novel schema-guided approach to wrapper generation. We provide a user-friendly interface that allows users to define the schema of the data to be extracted, and specifies mappings from a HTML page to the target schema. Based on the mappings, the system can automatically generate an extraction rule to extract data from the page. Our approach to wrapper generation can significantly reduce the work of human beings in this process. And the user never have to deal with the internal extraction rule, or even familiarity with the details of HTML.

1 Introduction

The World Wide Web has become one of the most important connections to various information sources. A large proportion of the Web data is embedded in HTML documents. This language serves the visual presentation of data in browsers, not for automated, computer-assisted information management systems. If data from different sources needs to be integrated, it is necessary to develop special and often complex programs to extract data from Web pages. To achieve this goal, people have developed w*rappers* [6], which are specialised programs that can automatically extract data from Web pages and convert the information into a structured format. The main issues of wrapper generation include (1) to identify semantics of the data contained in an HTML document and, (2) to establish the mappings between its structure and its semantics.

Different methods have been proposed to automate the wrapper-generation process. These works can be classified into three categories [2]: manual wrapper programming languages [6], machine learning approaches [1, 4, 7, 8] and supervised interactive wrapper generation [2, 10, 13]. In the first category, wrappers are programmed manually so that it is difficult to use by layperson. The second approaches have drawbacks of limited expressive power and the large number of required example pages. The last category includes XWRAP [10] and LIXTO [2]. XWRAP makes some progresses in user-friendly interactive method. But it does not appropriately take into account the user's view of HTML pages and possible changes of them. And LIXTO differs from the internal extraction rule and user defined schema. Under the user defined target data schema, we used XQuery[17] expression for representing extraction rule internally, which provides more powerful and precise ability for Web wrappers.

In this paper, we propose a novel schema-guided approach to wrapper generation. We provide a user-friendly interface that allows users to define the schema of the user target data, and specify mappings (see Section 3.2) from an HTML page to the target schema. Since different user may perhaps want different data from a Web page, so user defined schema can help a user expresses her target more precisely. Meanwhile, while building the mappings from the HTML page to the target schema, the user can also add additional conditions to make higher accurate, e.g. annotations of data items (see Section 3.2.1), or context conditions of the data items, and so on. All of these conditions will be recorded as predicates (XPath [16]) in the mappings (see Section 3.2.2). Based on the mappings, the system can automatically generate a wrapper to extract data from the page. The internal XQuery [17] based extraction rule is invisible to the user. After the user generate an initial wrapper, if she is not satisfied with the extracted results, she can select more instances (*Rule Refining*) and the system will automatically merge the initial and the new extraction rule. Intensive experiments on real-world Web pages show that this approach to wrapper generation can significantly reduce the work of human beings in this process and get satisfactory precisions. What's more, ordinary users who are not familiar with wrappers can also easily master the procedure of wrapper generator with SG-WRAP.

The rest of this paper is organized as follows. Section 2 describes the architecture of our system. Section 3 presents our approach of schema-guided wrapper generation. Section 4 provides the discussion of related works. In Section 5 we conclude the paper and discuss future research directions.

2 System Architecture

Figure 1 depicts the architecture of our SG-WRAP [11, 12] system. The system consists of five major components: *Preprocessor*, *Schema Acquirer, Rule Generator, Rule Refiner* and *Wrapper Generator*.

Preprocessor is responsible for setting up the environment for the system. It fetches the Web page using the URL given by the user. The fetched HTML page is displayed before user in the browser for the next steps.

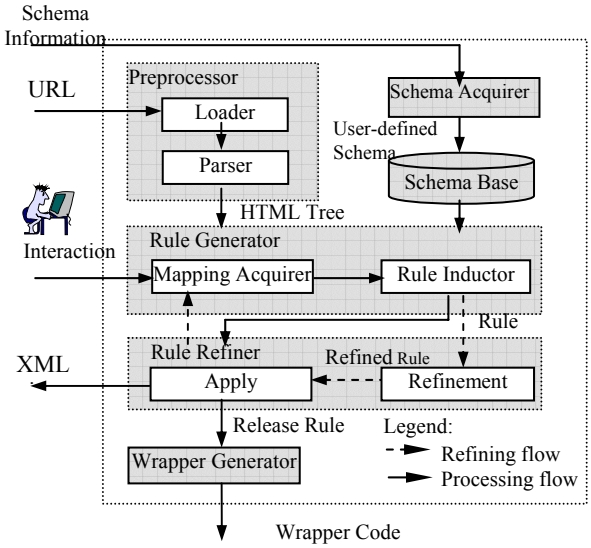

Figure 1: SG-WRAP: The system Architecture

Another input of the system is a user-defined schema for extracted data, which is obtained by the *Schema Acquirer*. A user defined schema can be saved in a Schema Base to be used later, or shared for other sources from which the same kinds of data to be extracted.

Rule Generator generates data extraction rule by an induction algorithm on the user assigned mappings between schema elements and HTML document data nodes, which takes the list of mapping instances as input and returns a candidate rule by incorporating the similar mapping rule instances into a new extraction rule.

Rule Refiner generates an XML document by applying the induced rule on the input page. From the displayed document, a user determines whether the current extraction rule correctly extracts required data from the source page. If not, she can identify more mapping instances. The induction process will repeat and the Refiner will merge the previously generated rules with the refined rules. The system will display a new version of result to the user for checking. This refining process continues until the user is satisfied with the data extracted.

After all the steps are taken, *Wrapper Generator* materializes the extraction rule into Java program and outputs it for repeatedly usages.

3 Supervised Visual Wrapper Generation

In this section we describe our approach to generating wrapper rules to extract data from HTML pages. The main idea of the approach is the following. A user defines the structure of her target information by providing an XML schema in the form of a DTD. Given an HTML page, by using a GUI toolkit, the user creates mappings from useful values in the HTML page to the corresponding schema elements. Internally the system parses the HTML page into a tree, and generates the corresponding mappings from the HTML tree to the schema tree. Using these mappings the system can generate a tree pattern and output an extraction rule. This section depicts the details of the process.

```
<!ELEMENT BookList (Book+)>
<!ELEMENT Book(Title,PubDate,Price,Authors)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT PubDate (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Authors (Author+)>
<!ELEMENT Author (#PCDATA)>
```

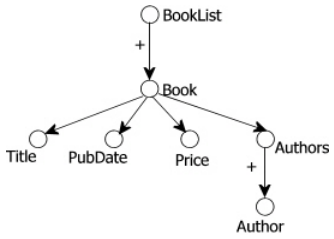

Figure 2: Target DTD schema.

3.1 Specifying Mappings Using GUI

To extract data from a set of HTML pages sharing the same layout, a user first defines the structure of the target data by providing an XML schema tree in the form of a DTD. She inputs the schema by either (1) providing a DTD file; or (2) building the schema tree using the toolkit provided by the system; or (3) modifying an existing schema to meet her own requirements. The schema reflects the user's view of the XML document to be generated. The positions of elements in the schema tell the system how to organize the extracted data. Given an HTML page, the goal of the system is to generate a rule to extract data from the page and generate a document conforming to the schema. In addition, the rule should be able to extract data from other pages that share the same structure as the given page.

As an example, suppose the user wants to extract book information from the book pages at www.buy.com A sample page is shown in Figure 3. The user specifies the following DTD, whose tree representation is shown in Figure 2.

After the user provides an HTML page (e.g., by typing in an URL), she can use a GUI interface (shown in Figure 3) to create mappings from data items on the HTML page to the corresponding schema elements. The left-hand side displays the HTML page, and the right-hand side presents the tree structure of the DTD schema. The user first uses the mouse to highlight the region of an interesting data value, such as a book title, author, and price. (In Figure 3, she has highlighted a book author *David Flanagan* in the HTML page.) For such a highlighted value, the system displays a list of candidate *annotations* in the page. An *annotation* is the contents of a text node or several text nodes in the page that could possibly describe the content of this value. For this book author, the system suggests the string "*Author*" before *David Flanagan* as a candidate annotation. The user can either choose one of the system-suggested annotations, or select an annotation by manually

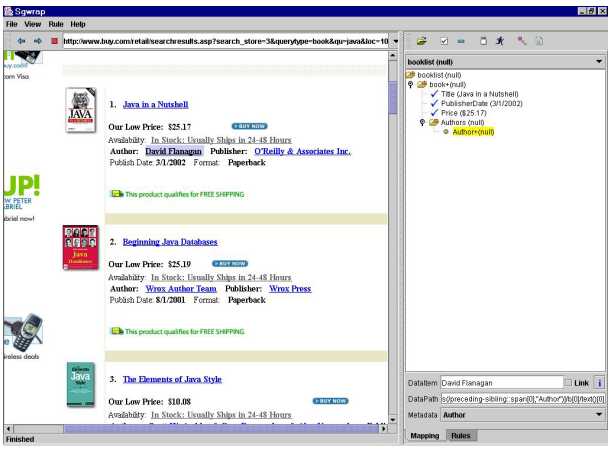

Figure 3 GUI

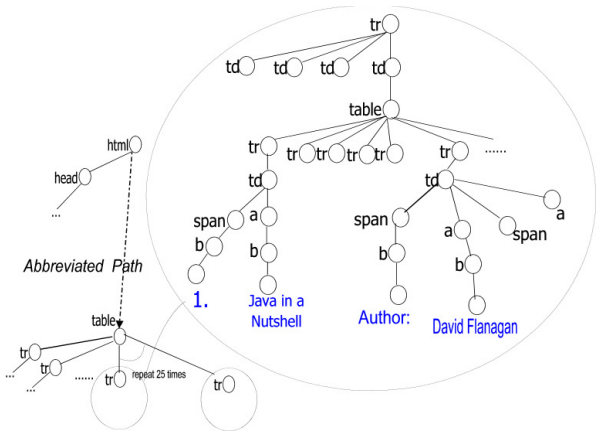

Figure 4 DOM Tree

highlighting a string on the page. Table 1 shows the final annotations of a few data values. Notice that the annotation of a value could be empty.

After selecting the annotation for a data item on the HTML page, the user highlights a corresponding element in the schema tree, and a mapping is created from the data value to this schema element. In Figure 3, the user has selected a schema element `author` for the data element *David Flanagan*. Note that while highlighting HTML data elements, the user needs to make sure that these elements belong to the same data entity, e.g., the price and title of the *same* book. Then the system can generate a rule to extract the values of the book.

Table 1: Annotations for HTML data values

Data values in HTML page	Annotations
David Flanagan	Author
O'Reilly & Associates Inc.	Publisher
3/1/2002	Publish Date
$25.17	Our Low Price
Java in a Nutshell	None

3.2 Generating an Extraction Rule

After the user creates these mappings, the system generates an extraction rule. The rule can also be used to extract data from other book pages with the same structure. The system generates the rule in three steps: (1) parsing and representing the HTML page as a tree; (2) generating mappings from the HTML tree and the schema tree; and (3) computing a *tree pattern* from the HTML tree to generate an extraction rule. Here we explain these steps in detail. Notice these steps are executed internally in the system, and they are transparent to the user.

3.2.1 Step 1: Creating an HTML Tree

The HTML document is parsed to a tree representation. A non-leaf node corresponds to a pair of HTML tags, such as "<tr>…</tr>". Nodes are ordered according to their occurrence sequence in the HTML page. Each leaf node is either of type *PCDATA*, or a single tag such as "<tr>".

Figure 4 shows the parsed tree of the HTML page given in Figure 3. This tree representation follows the specification of the Document Object Model (DOM) [14]. A primary advantage of this representation is that we can uniquely define an element in the tree with an XPath expression. For example, the following XPath expression

/body/table[0]/tr[2]/td[0]/table[1]/tr[0]/td[2]/table[0]/tr[1]/td[0]/table[0]/tr[5]/td[3]/table[0]/tr[5]/td[0]/a[0]/b[0]/text()

identifies the data value *David Flanagan* in the HTML page. In the expression, method "text()" selects all the text of the children of the context node. (See XPath [16] for more detail.)

Suggesting Annotations: After the user highlights a data value in the HTML page, the system suggests candidate annotations for this value using several heuristics. For example, we can use the surrounding strings of the data value, or if the data value appears in a table, we can suggest strings in the first row of the table. According to the HTML specification [15], we may use the tag <th> to show the difference between the table head and the table content. In the following table in an HTML page, the string *Manufacturer* is a possible annotation of data values *Dell* and *Apple*. The string *Price* can be a candidate annotation of data values *$1,380* and *$1,520*.

Manufacturer	Price
Dell	$1,380
Apple	$1,520

After the user chooses an (possibly empty) annotation for an HTML data item, the system adds the annotation to the path of the item with the function "contains" provided by XPath. This method is used to mark the annotation of the data value. The first parameter of "contains" shows the location of annotation corresponding to the data value and the second one is the annotation string.

```
{LET $blist := document ($d)
    RETURN
    <Booklist>
      {FOR $b IN
          $blist/body/table[0]/tr[2]/td[0]/table[1]/tr[0]/td[2]
          table[0]/tr[0]/td[0]/table[0]/tr
       RETURN
       <Book>
         {FOR $t IN
           $b/td[3]/table[0]/tr[1]/td[0]a[0]/b[0]/text()[0]
            RETURN<Title> $t</Title>
         }
         {LET $p = $b/tr[3]/td[0]/span[0][contains(contains(/parent
           /parent/preceding-sibling/b::text()[0],"Our Low Price")]
           /b[0]/text()[0]
            RETURN<Price> $p </Price>
         }
         { LET  $d = $b/tr[6]/td[0]/span[0]/b[0][contains(/parent/
           prediding-sibling/b::text()[0], "Publish Date")]/text()
            RETURN<PubDate>$d<PubDate>
        }
        {FOR $auths IN $Book
           RETURN
          <Authors>{
              FOR $a IN
              $auths/tr[5]/td[0]/a[contains(/parent/parent/parent/
              preceding-sibling/nobr/b::text()[0],"Author")]/b[0]
              /text()
              RETURN<Author> $a</Author>
          }
        }
        </Authors>
      </book>
     }
  </booklist>
}
```

Figure 5: An extraction rule.

3.2.2 Step 2: Computing Mappings from HTML Tree to Schema Tree

For each mapping from a data value to a schema element, the system computes the corresponding *internal mapping* from the HTML tree node to the schema element. Formally, each internal mapping is a tuple in the form *M(D, HP, SP)*, in which:

- *D*: is the HTML data value.
- *HP*: is the XPath expression of *D* in the HTML tree (meaning "HTML path").
- *SP*: is the path of the corresponding leaf node in the schema tree (meaning "schema path").

The following are the internal mappings for those mappings specified by the user in Figure 3.

M1(*D*: "Java in a Nutshell",
HP: …/td[3]/table[0]/tr[1]/td[0]/a[0]/b[0]/text()[0],
SP: BookList/Book/Title).

M2(*D*: "3/1/2002",
HP: …/td[3]/table[0]/tr[6]/td[0]/span[0]/b[0][contains(
/parent/preceding-sibling::text()[0], "Publish Date")]
/text(),
SP: BookList/Book/PubDate).

M3(*D*: "$25.17",
HP: …/td[3]/table[0]/tr[3]/td[0]/span[0]/b[0][contains(/
parent/parent/preceding-sibling/b::text()[0], "Our Low Price")]/text();
SP: BookList/Book/Price).

M4(*D*: "David Flanagan",
HP: …/td[3]/table[0]/tr[5]/td[0]/a[0]/b[0][contains(/par
ent/parent/parent/preceding-
sibling/nobr/b::text()[0], "Author:")]/text();
SP: BookList/Book/Authors/Author).

```
Global variable: Mappings MS;
Procedure genPattern(DTD element e, HTMLTree H)
{
    G = {mappings in MS whose SP includes e};
    P = common path of mappings in G;
    If (e is marked with '*' or '+' in the DTD)
       P = generalizePath(P); /* generalize the common
            path in order to extract similar data values */
    st = subtree identified by the P;
    sp = subpattern identified by the P;
    i = 0;
    for each child element c of e do {
       /* generate the sub pattern for each c*/
       sp[i++] = genPattern (c, st); // recursive call
    }
    R = construct the rule using P and sp;
    return R;
}
```

Figure 6: Algorithm genPattern()

Note that each HP path is not just a sequence of HTML tags. Instead, it is an XPath expression containing more information such as the annotation of the value and the position of the annotation in the HTML tree. The preceding-sibling axis [17] contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty. It is an axis of XPath.

3.2.3 Step 3: Computing an Extraction Rule

After computing the internal mappings from the HTML tree to the target DTD tree, the system computes an extraction rule by generating a tree pattern from the tuples in the mappings. The computed rule is shown in Figure 5.

This rule is an FLWR expression of XQuery [17]. By applying this expression on the HTML page, we can generate an XML document of the DTD. In general, in an extraction rule:

- A schema element marked with symbol "+" or "*" (e.g., `BookList`) corresponds to a clause of "`FOR … RETURN …`".
- Any other element (e.g., `Author`, `Price`, etc.) corresponds to a clause of "`LET … RETURN …`".

The structure of the rule is based on the DTD schema. For each `LET` or `FOR` clause, the system fills in the appropriate XPath on the HTML tree based on the internal mappings.

The extraction rule is generated by calling a recursive function, *genPattern()*. This function takes a schema element *e* as an input parameter, and generates a

corresponding XPath pattern in the HTML tree. This pattern is used in the clause of this element *e* in the final extraction rule, and it extracts data values to construct a subtree of this schema element. The system generates an extraction rule by calling this function passing the DTD root element.

Figure 6 shows the algorithm genPattern(). This algorithm starts from the root of the DTD tree and finally computes the extraction rule by recursively calling itself on the children of the current DTD element. It has two input parameters: (1) a schema element and (2) the current HTML subtree corresponding to the schema element. It outputs a XPath pattern for this schema element. The algorithm first finds all the mappings whose *SP* paths contain this element. Then it computes a common path of the subtrees in the HTML tree that can include these mappings. A common path is the exactly same parts of the XPath expression. Next, if the element is marked by symbol "*" or "+" in the schema tree, there may be multiple subtrees that contain the input mappings of the current instance, but we do not know their exact paths in the *H* tree. Thus the algorithm searches similar subtrees at the same level of this corresponding node. To represent all the matched subtrees, the algorithm tentatively generalizes the original common path by removing predicates at one or more steps. At last, the generalized common path is used to generate the rule for this element. If the element is not a leaf, the algorithm calls itself recursively for each of its child elements. The rules returned are added to the current rule as subrules.

Let us use the schema element `BookList` in Figure 3 to illustrate how the rule in Figure 5 is generated by calling the function genPattern() from the DTD root element `BookList`. The system first considers the four internal mappings M1, M2, M3, and M4 to find whose SP paths all include this element. Their common path is

/body/table[0]/tr[2]/td[0]/table[1]/tr[0]/td[2]/table [0]/tr[1]/td[0]/table[0]/tr[5]/td[3]/table[0]

Since the element of `BookList` has a child element, `Book`, the algorithm calls itself recursively using with the `Book` element and the subtree corresponding to the element of `BookList`. This element is marked with "+" in the DTD schema, thus the generalizePath(P) is called and we generalize the path to this subtree by searching its similar sibling subtrees. The following is the result of generalizePath(P), in which the "**tr**" in bold face is the generalized node.

*body/table[0]/tr[2]/td[0]/table[1]/tr[0]/td[2]/table [0]/tr[1]/td[0]/table[0]/**tr**/td[3]/table[0]*

The algorithm calls itself recursively for each of the four children of the element `Book`. It computes one rule for each of them and adds the new rule as a subrule. If the schema element does not have a corresponding HTML node, the algorithm will not find mappings whose SP paths contain this element. Thus the computed rule will not contain this schema element.

Finally the rule in Figure 5 is generated. The final rule is a source-specific program that can be used to extract data from other pages with the similar structure as the given page. For example, the rule generated from the sample page in Figure 3 should also be used to extract data from other book pages of the web site with the same structure.

3.3 Refine Wrappers

In the previous sections, we have induced a tree pattern for a single page from a single instance. While it's possible that the user happens select a bad instance. Let's look at Figure 5, e.g. we find that the selected instance should contains *book title*, *price* and *authors*. But a user may happen to select an instance only contains information of *title* and *price*. In other word, the user selected an instance containing insufficient information. Thus the finally induced extraction rule may fail to extract the information of *authors* of from other parts of the page.

What's more, if we want to apply the wrapper on a set of pages with the similar structures, e.g. a set of pages from the same search engine of www.buy.com, we may possibly find that the wrapper fail to correctly extract all the data items, a few data items are perhaps missed because of some tiny difference in structures comparing with the common structure. For instance, we find that the data items of *authors* contains a hyperlink on them in this example page, but it's quite possible that in another page the data items of *authors* do not contains hyperlink because of the web site hasn't store the information of this author.

The *Rule Refiner* of SG-WRAP also provides an interface for testing the generated wrapper. The user can apply the wrapper on the example pages and examine that if the results are satisfactory, otherwise, she can take the following step for refining wrappers.

To deal with the first case, the user selected a bad instance, with the module of *Rule Refiner* (see Section 2), if a user finds that the results are not satisfactory, she can select more instances and induce extraction rules from them. Then the SGWRAP system can automatically integrate these extraction rules. Since our extraction rule uses XQuery expression, it's easy to integrate two extraction rules. The process is the following:

- If the rule from new instances contains new predicates for a certain data item, the predicates are added into the extraction rule.
- If the new rule contains a new path to a data item, the path is added into the extraction rule.

Also, to perfectly extract data items from a set of pages with similar structure, a user may need to select instances from another page, and then the system can automatically integrates the extraction rules.

4 Related Work

Several wrapper generation methods have been discussed in the literatures. TSIMMIS [6] introduced a logical template-based approach to constructing wrappers by example but the wrapper output has to obey the document structure. The project of ARIADNE [7] at University of Washington presents a semi-automatic

wrapper construction method that treats documents as token flow and takes LEX and YACC to deal with the semantic units and the nested structure of Web page. Some machine learning approaches are introduced to data extraction. STALKER [8] specializes general SkipTo sequence patterns based on labeled HTML pages. Kushmerick et al. [9] create robust wrappers which base on predefined extractors; their visual support tool WIEN receives a set of training pages. Their approaches do not use the HTML tree so that they have restricted capabilities to deal with HTML. In general, machine learning approaches have drawbacks of limited expressive power and the large number of required example pages. Recently, supervised interactive wrapper generation approaches are developed. W4F [13] developed a toolkit to help the developers to generate wrappers and used Nested String Language(NSL) to encode the information extraction rules. However, the developer requires expertise with both HEL and HTML to program extraction rule manually. XWRAP [10] uses a procedural rule system and provides limited expressive power for pattern definition. The user cannot label the desired data items directly on the browser-displayed document. LIXTO [2] proposed a visual method for data extraction. It provides visual facilities for imposing external or internal conditions to a pattern. However, these conditions are not automatically added into a pattern; instead, it is specified by the user. Furthermore, their endeavors do not appropriately take the user's view of HTML pages and possible changes of them into account.

5 Conclusion and Future Work

In this paper, we propose a schema-guided supervised visual wrapper generation approach for Web-data extraction. The toolkit SGWRAP was developed to semi-automatically generate Web wrapper particularly for extracting locally well-formatted data from Web HTML pages. In contrast to extracting small locally well-formatted data, large text information elements in Web pages pose several new problems. In order to automatically generate the wrappers capable of extracting large information elements such as news story documents from Web pages, SG-WRAP should be extended in the future.

6 Acknowledgements

This research was partially supported by the grants from 863 High Technology Foundation of China under grant number 2002AA116030, the Natural Science Foundation of China (NSFC) under grant number 60073014, 60273018, the Key Project of Chinese Ministry of Education (No.03044) and the Excellent Young Teachers Program of M0E，P.R.C (EYTP).

7 References

[1] Ashish N, Knoblock C A. Wrapper generation for semi-structured Internet sources. SIGMOD Record, 1997, 26(4): 8-15.

[2] Baumgartner R, Flesca S, Gottlob G..Visual Web Information Extraction with Lixto. In Proceedings of the Very Large Data Bases; 2001, 119-128.

[3] Brin S. Extracting patterns and relations from the world wide web. In International WebDB Workshop, Valencia, Spain, pages 172-183, 1998.

[4] Doorenbos R, Etsionoi O, Weld D S. A scalable comparison-shopping agent for the world-wide-web. In Proceedings of the First International Conference on Autonomous Agents, 1997, 39-48.

[5] Gupta A., Harinarayan V., Quass D., and Rajaraman A. Method and apparatus for structuring the querying and interpretation of semistructured information. United States Patent number 5,826,258, 1998.

[6] Hammer J, Brenning M, Garcia-Molina H, Nestorov S, VassalosV, Yerneni R. Template-based wrappers in the TSIMMIS system. In Proceedings of ACM SIGMOD Conference, 1997, 532-535.

[7] A. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach. Bulletin of the IEEE Computer Society Technial Committee on Data Engineering, 1999

[8] Knoblock C A, Lerman K, Minton S, Muslea I. Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2000, 23(4): 33-41.

[9] Kushmerick N, Weil D, Doorenbos R. Wrapper induction for information extraction. In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 1997, 729-735.

[10] Liu L, Pu C, Han W. XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources. In Proceedings of ICDE, 2000, 611-621.

[11] Meng X F, Lu H J, Wang H Y, Gu M Z. SG-WRAP: A Schema-Guided Wrapper Generator. Demonstration in ICDE, 2002, 331-332.

[12] Meng X F, Lu H J, Wang H Y, Gu M Z. Schema-Guided Data Extraction from the Web. Journal of Computer Science and Technology (JCST), 2002,17(4).

[13] Sahuguet A, Azavant F. Building Light-Weight Wrappers for Legacy Web Data-Sources Using W4F. In Proceedings of VLDB, 1999, 738-741.

[14] DOM Document Object Model (DOM) Level 2 Core Specification http://www.w3.org/TR/DOM-Level-2-Core

[15] HTML 4.01 Specification, http://www.w3.org/TR/html401/

[16] XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/

[17] XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/