# XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources

Ling Liu, Calton Pu, Wei Han

Georgia Institute of Technology
College of Computing, Atlanta, Georgia 30332-0280
{lingliu,calton,weihan}@cc.gatech.edu

**Abstract.** *This paper describes the methodology and the software development of XWRAP, an XML-enabled wrapper construction system for semi-automatic generation of wrapper programs. By XML-enabled we mean that the metadata about information content that are implicit in the original web pages will be extracted and encoded explicitly as XML tags in the wrapped documents. In addition, the query-based content filtering process is performed against the XML documents. The XWRAP wrapper generation framework has three distinct features. First, it explicitly separates tasks of building wrappers that are specific to a Web source from the tasks that are repetitive for any source, and uses a component library to provide basic building blocks for wrapper programs. Second, it provides a user-friendly interface program to allow wrapper developers to generate their wrapper code with a few mouse clicks. Third and most importantly, we introduce and develop a two-phase code generation framework. The first phase utilizes an interactive interface facility to encode the source-specific metadata knowledge identified by individual wrapper developers as declarative information extraction rules. The second phase combines the information extraction rules generated at the first phase with the XWRAP component library to construct an executable wrapper program for the given web source. We report the initial experiments on performance of the XWRAP code generation system and the wrapper programs generated by XWRAP.*

## 1. Introduction

The extraordinary growth of the Internet and World Wide Web has been fueled by the ability it gives content providers to easily and cheaply publish and distribute electronic documents. Companies create web sites to make available their online catalogs, annual reports, marketing brochures, product specifications. Government agencies create web sites to publish new regulations, tax forms, and service information. Independent organizations create web sites to make available recent research results. Individuals create web sites dedicated to their professional interest and hobbies. This brings good news and bad news.

The good news is that the bulk of useful and valuable HTML-based Web information is designed and published for human browsing. The bad news is that these "human-oriented" HTML pages are difficult for programs to parse and capture. In addition, most of the web information sources are created and maintained autonomously, and each offers services independently. A popular approach to address the problem of data integration on the Web is to write *wrappers*[1] to encapsulate the heterogeneity in accessing diverse data sources. For instance, the most recent generation of information mediator systems (e.g., Ariadne [10], CQ [13, 14], Internet Softbots [12], TSIMMIS [7, 8]) all include a pre-wrapped set of web sources to be accessed via database-like queries. However, developing and maintaining wrappers by hand turned out to be labor intensive and error-prone.

In this paper, we propose an adaptive approach to build an interactive system for semi-automatic construction of wrappers for Web information sources, called XWRAP. The goal of our work can be informally stated as the transformation of "difficult" HTML input into "program-friendly" XML output, which can be parsed and understood by sophisticated query services, mediator-based information systems, and agent-based systems. A main technical challenge is to discover boundaries of meaningful objects (such as regions and semantic tokens) in a Web document, to distinguish the information content from their metadata description, and to recognize and encode the metadata explicitly in the XML output.

[1] Wrappers are software programs that can transform data from a less structured representation into a more structured representation.

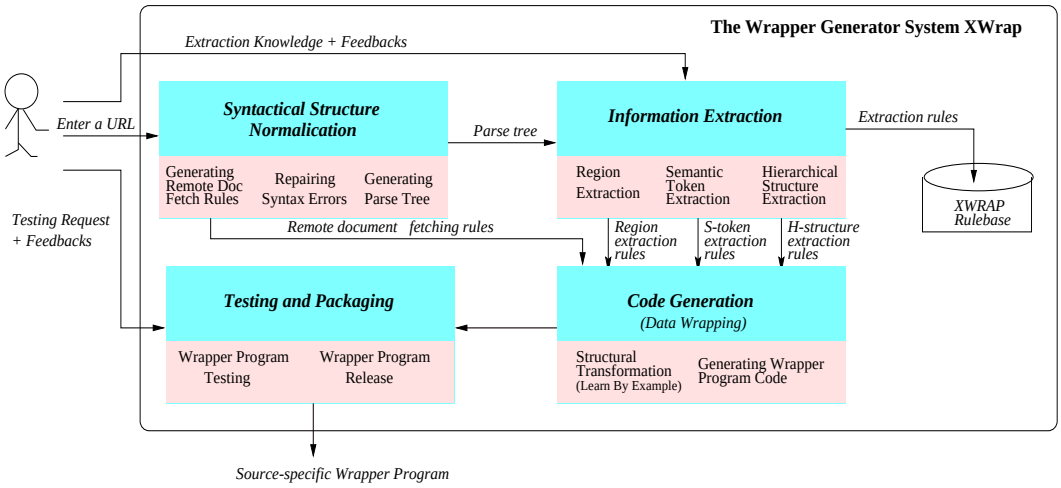


Figure 1: XWRAP system architecture for data wrapping

This is not the first time the problem of information extraction from a Web document has been addressed. [3, 9] discover object boundaries manually. They first examine the documents and find the HTML tags that separate the objects of interest, and then write a program to separate the object regions. [1, 16, 2, 10, 11, 12, 17] separate object regions with some degree of automation. Their approaches rely primarily on the use of syntactic knowledge, such as specific HTML tags, to identify object boundaries.

The first main contribution of the XWRAP approach is a set of interactive mechanisms and heuristics for generating information extraction rules with a few clicks. The second contribution is the two-phase code generation approach for generating executable wrapper programs. The first phase utilizes an interactive interface facility that communicates with the wrapper developer and generates information extraction rules by encoding the source-specific metadata knowledge identified by the individual wrapper developer. In contrast, most of the existing approaches require the wrapper developers to write information extraction rules by hand using a domain-specific language. The second phase utilizes the information extraction rules generated at the first phase and the XWRAP component library to construct an executable wrapper program for the given web source. The two-phase code generation approach presents a number of advantages over existing approaches. First, it provides a user-friendly interface program to allow users to generate their information extraction rules with a few clicks. Second, it provides a clean separation of the information extraction semantics from the generation of procedural wrapper programs (e.g., Java code). Such separation allows new extraction rules to be incorporated into a wrapper program incrementally. Third, it facilitates the use of the micro-feedback approach to revisit and tune the wrapper programs at run time.

## 2. The Design Framework

The architecture of XWRAP for data wrapping consists of four components - Syntactical Structure Normalization, Information Extraction, Code Generation, Program Testing and Packaging. Figure 1 illustrates how the wrapper generation process would work in the context of data wrapping scenario.

**Syntactical Structure Normalization** is the first component and also called Syntactical Normalizer, which prepares and sets up the environment for information extraction process by performing the following three tasks. First, the syntactical normalizer accepts an URL selected and entered by the XWRAP user, issues an HTTP request to the remote server identified by the given URL, and fetches the corresponding web document (or so called page object). This page object is used as a sample for XWRAP to interact with the user to learn and derive the important information extraction rules. Second, it cleans up bad HTML tags and syntactical errors [15, 18]. Third, it transforms the retrieved page object into a parse tree or so-called syntactic token tree.

**Information Extraction** is the second component, which is responsible for deriving extraction rules that use declarative specification to describe how to extract information content of interest from its HTML formatting. XWRAP performs the information extraction task in three steps - (1) identifying interesting regions in the retrieved document, (2) identifying the important semantic tokens and their logical paths and node positions in the parse tree, and (3) identifying the useful hierarchical structures of the retrieved document. Each step results in a set of extraction rules specified in declarative languages.

**Code Generation** is the third component, which generates the wrapper program code through applying the three sets of information extraction rules produced in the second step. An essential technique in our implementation is the smart encoding of the semantic knowledge represented in the form of declarative extraction rules and XML-template format (see Section ). The code generator interprets the XML-template rules by linking each executable component with each type of rules. We found that XML gives us great extensibility to add more types of rules seamlessnessly. As a byproduct, the code generator also produces the XML representation for the retrieved sample page object.

**Testing and Packing** is the fourth component and the final phase of the data wrapping process. The toolkit user may enter a set of alternative URLs of the same web source to debug the wrapper program generated by running the XWRAP automated testing module. For each URL entered for testing purpose, the testing module will automatically go through the syntactic structure normalization and information extraction steps to check if new extraction rules or updates to the existing extraction rules are derived. In addition, the test-monitoring window will pop up to allow the user to browse the test report. Whenever an update to any of the three sets of the extraction rules occurs, the testing module will run the code generation to generate the new version of the wrapper program. Once the user is satisfied with the test results, he or she may click the release button (see Figure 7) to obtain the release version of the wrapper program, including assigning the version release number, packaging the wrapper program with application plug-ins and user manual into a compressed tar file.

Due to the space restriction, in the subsequent sections we focus our discussion primarily on information extraction component of the XWRAP, and provide a walkthrough example to illustrate how the three sets of information extraction rules are identified, captured, and specified.

## 3. Information Extraction

The main task of the information extraction component is to explore and specify the structure of the retrieved document (page object) in a declarative extraction rule language. For an HTML document, the information extraction phase takes as input a parse tree generated by the syntactical normalizer. It first interacts with the user to identify the semantic tokens (a group of syntactic tokens that logically belong together) and the important hierarchical structure. Then it annotates the tree nodes with semantic tokens in comma-delimited format and nesting hierarchy in context-free grammar. More concretely, the information extraction process involves three steps: (1) Identifying regions of interest on a page; (2) Identifying semantic tokens of interest on a page; and (3) Determining the nesting hierarchy for the content presentation of a page. Each of the three steps generates a set of extractions rules to be used by the code generation phase to generate wrapper program code.

### 3.1. Preprocessing

Before the information extraction process begins, the Syntactical Structure Normalization is performed. It fetches the remote documents and repairs the bad syntax. The clean HTML document is fed to a source-language-compliant tree parser, which parses the block character by character, carving the source document into a sequence of atomic units, called *syntactic tokens*. Each token identified represents a sequence of characters that can be treated as a single syntactic entity. The tree structure generated in this step has each node representing a syntactic token, and each tag node such as `TR` represents a pair of HTML tags: a beginning tag `<TR>` and an end tag `</TR>`. All non-leaf nodes are tags and all leaf nodes are text strings, each in between a pair of tags. Different languages may define what is called a token differently. For HTML pages, the usual tokens are paired HTML tags (e.g., `<TR>`, `</TR>`), singular HTML tags (e.g., `<BR>`, `<P>`), semantic token names, and semantic token values.

**Example 1** Consider the weather report page for Savannah, GA at the national weather service site (see Figure 2), and a fragment of HTML document for this paper in Figure 3. Figure 4 shows a portion of the HTML tree structure, corresponding to the above HTML fragment, which is generated by running a

```
<TABLE><TR><TD COLSPAN=3><H3><FONT FACE="Arial, Helvetica">Maximum and Minimum Temperatures</FONT>
</H3> </TD></TR><TR><TD ALIGN=CENTER BGCOLOR="#FFFFFF"><B><FONT COLOR="#0000A0"><FONT FACE=
"Arial,Helvetica">Maximum<BR>Temperature<BR>F(C)</FONT></FONT></B></TD><TD ALIGN=CENTER BGCOLOR=
"#FFFFFF"><B><FONT COLOR="#0000A0"><FONT FACE="Arial,Helvetica">Minimum<BR>Temperature<BR>F(C)
</FONT></FONT></B></TD><TD></TD></TR><TR><TD ALIGN=CENTER><FONT FACE="Arial,Helvetica">82.0(27.8)
</FONT></TD><TD ALIGN=CENTER><FONT FACE="Arial, Helvetica">62.1(16.7)</FONT></TD><TD><FONT FACE=
"Arial, Helvetica">In the <B>6 hours</B> preceding Oct 29, 1998 - 06:53 PM EST / 1998.10.29 2353
UTC</FONT></TD></TR><TR><TD ALIGN=CENTER><FONT FACE="Arial,Helvetica">80.1(26.7)</FONT></TD>
<TD ALIGN=CENTER><FONT FACE="Arial, Helvetica">45.0(7.2)</FONT></TD><TD><FONT FACE="Arial,
Helvetica">In the <B>24 hours</B> preceding Oct 28, 1998 - 11:53 PM EST / 1998.10.28 0453 UTC</FONT>
</TD></TR><TR><TD COLSPAN=3><HR SIZE=1 NOSHADE WIDTH="100%"></TD></TR></TABLE> ......
```

Figure 3: An HTML fragment of the weather report page at nws.noaa.gov site

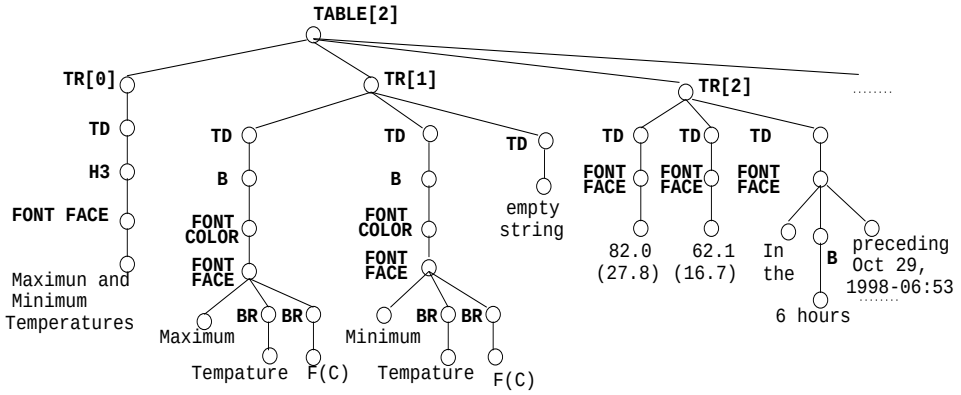


Figure 4: A fragment of the HTML tree for the Savannah weather report page

HTML-compliant tree parser on the Savannah weather source page. In this portion of the HTML tree, we have the following six types of tag nodes: `TABLE`, `TR`, `TD`, `B`, `H3`, `FONT`, and a number of semantic token nodes at leaf node level, such as `Maximum Temperature`, `Minimum Temperature`, `84.9(29.4)`, `64.0(17.8)`, etc.

XWRAP defines a set of tree node manipulation functions for each tree node object. We use dot notation convention to represent the node path. A single-dot expression such as nodeA.nodeB refers to the parent-child relationship and a double-dot such as nodeA..nodeB refers to the ancestor-descendent relationship between nodeA and nodeB.

### 3.2. Region Extraction

Region extraction is performed via an interactive interface, which lets the XWRAP user guide the identification of important regions in the source document, including table regions, paragraph regions, bullet-list regions, etc. The output of this step is the set of region extraction rules that identify regions of interest from the parse tree.

In the first prototype of XWRAP, the region extractor begins by asking the user to highlight the tree node that is the start tag of an important element. Then the region extractor will look for the corresponding end tag, identify and highlight the entire region. In addition, the region extractor computes the type and the number of sub-regions and derives the set of region extraction rules that capture and describe the structure layout of the region. For each type of region, such as the table region, the paragraph region, the text section region, and the bullet list region, a special set of extraction rules are used.

For example, for regions of the type `TABLE`, Figure 5 shows the set of rules that will be derived and finalized through interactions with the user. The rule `Tree_Path` specifies how to find the path of the table node. The rule `Table_Area` finds the number of rows

```
Region_Extraction_Rules(String source_name)::
    Tree_Path(String node_id, String node_path){
        setTableNode = node_id;
        node_path = getNodePath(node_id); }

    Table_Area(String node_id, String TN, String CN, Integer rowMax, Integer colMax){
        setRowTag(node_id) = ?TN;
        setColTag(node_id) = ?CN;
        rowMax = getNumOfRows(node_id);
        colMax = getNumOfCols(node_id); }

    Effective_Area(String node_id, String rowSI, String rowEI, String colSI, String colEI){
        setRowStartIndex(node_id) = ?rowSI;
        setRowEndIndex(node_id) = ?rowEI;
        setColStartIndex(node_id) = ?colSI;
        setColEndIndex(node_id) = ?colEI;
        getEffectiveArea(node_id); }

    Table_Style(String node_id){
        if (ElementType(child(child(node_id, 1), 1)) = 'Attribute'
            if ElementType(child(child(node_id, 1), 2)) = 'Attribute')
                setVertical(node_id) = 1, setHorizontal(node_id) = 0;
            else
                setHorizontal(node_id) = 1, setVertical(node_id) = 0; }

    getTableInfo(String node_id, String TNN, String TN, String TP){
        setTableNameNode(node_id) = TNN;
        TN = getTableName(TNN);
        TP = getNodePath(TNN); }
```

Figure 5: Extraction rules for a table region in an HTML page

and columns of the table. The rule `Effective_Area` defines the effective area of the table. An effective area is the sub-region in which the interesting rows and columns reside. By differentiating the effective area from a table region, it allows us, for example, to remove those rows that are designed solely for spacing purpose. The fourth rule `Table_Style` is designed for distinguishing vertical tables where the first column stands for a list of attribute names from horizontal tables where the first row stands for a list of attribute names. The last rule `getTableInfo` describes how to find the table name by giving the path and the node position in the parse tree.

**Example 2** Consider the weather report page for Savannah, GA at the national weather service site (see Figure 2), and a fragment of HTML parse tree as shown in Figure 4). To identify and locate the region of the table node TABLE[2], we apply the region extraction rules given in Figure 5 and obtain the following source-specific region extraction rules for extracting the region of the table node `TABLE[2]`.

1. By applying the first region extraction rule, XWRAP can identify the tree path for `TABLE[2]` to be `HTML.BODY.TABLE[0].TR[0].TD[4].TABLE[2]`.

2. To identify the table region, we first need the user to identify the row tag TR and the column tag TD of the given region of the TABLE[2] node. Based on the row tag and column tag, the region extractor may apply the second extraction to deduce that the table region of `TABLE[2]` consists of maximum 5 rows and maximum 3 columns.

3. The extraction rule `Effective_Area` will be used to determine the effective area of the table node `TABLE[2]`. It requires the user's input on the row start index `rowSI = 2`, the row end index `rowEI = 4`, the column start index `colSI = 1` and the column end index `colEI = 3`.

4. By applying the rule `Table_Style`, we can deduce that this table is a horizontal table, with the first row as the table schema.

5. To determine how to extract the table name node, XWRAP first infers the path expression

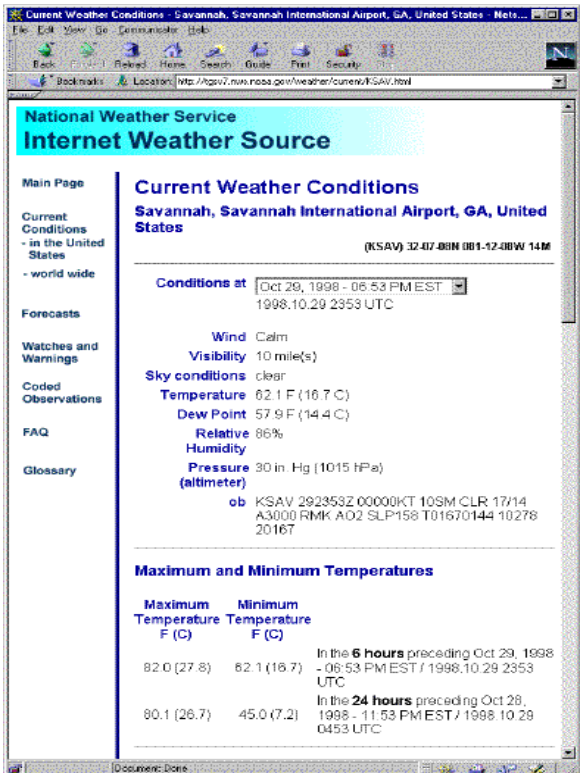


Figure 2: An example weather report page at the nws.noaa.gov site

for the table name node highlighted by the user. Then by applying the fifth region extraction rule `getTableInfo`, we can extract the table name. (see Section  for details on semantic token extraction).

The design of our region extraction rules is robust in the sense that all the important information (such as the number of tables in a page, the number of attributes in a table) will be computed at runtime. In addition, the region extraction rules are defined in a declarative language, independent of the implementation of the wrapper code. This higher level of abstraction allows the XWRAP wrappers to enjoy better adaptability to unexpected changes at the remote sources.

### 3.3. Semantic Token Extraction

The *semantic-token extractor* is an interactive program, which guides a wrapper developer to walk through the tree structure generated by the syntactic normalizer, and highlight the semantic tokens of interest in the source document page. The output of this step includes a set of semantic token extraction rules that can be used to locate and extract the semantic tokens of interest of the Web documents from the same web site, and a comma-delimited file[2], containing all the element type and element value pairs of interest. The first line of a comma-delimited file contains the name of the fields that denote the data. A special delimiter should separate both field names and the actual data fields. The XWRAP system supports a variety of delimiters such as a comma (,), a semicolon (;), or a pipe (|). To identify important semantic tokens, the S-token extractor examines successive tree nodes in the source page, starting from the first leaf node not yet grouped into a token. The S-token extractor may also be required to search many nodes beyond the next token in order to determine what the next token actually is.

**Example 3** Consider a fragment of the parse tree for the Savannah weather report page shown in Figure 4. From the region extraction step, we know that the leaf node name `Maximum and Minimum Temperatures` of the left most branch `TR[0]` is the heading of a table region denoted by the node `TABLE[2]`. Also based on the interaction with the user, we know that the leaf nodes of the subtree anchored at `TABLE[2].TR[1].TD[0]` should be treated as a semantic token with the concatenation of all three leaf node names, i.e., the string `Maximum Temperature F(C)`, as the token name; and the leaf nodes of the tree branch `TABLE[2].TR[2].TD[0]`, i.e., the string `84.9 (29.4)`, is the value of the corresponding semantic token. Thus a set of semantic token extraction rules can be derived for the rest of the subtrees anchored at `TR[3]` and `TR[4]`, utilizing the function `getStoken()`.

```
<ST_extract>
ST_extract(String ST_name[], String ST_val[][])
<!-- Start of the repetition -->
<? XG-Iteration-XG ``Start"?>
 <loop> integer row_i = 3, 4
  <loop> integer col_j = 0,1,2
   <rule_exp>
    extract ST_val[row_i,col_j] =
      ~TABLE[2].TR[row_i].TD[col_j].getStoken()
    where ~TABLE[2].TR[1].TD[col_j].getStoken()
          = ST_name[col_j];
   </rule_exp>
  </loop>
 </loop>
</ST_extract>
```

[2] A comma-delimited format is also called delimited text format. It is the lowest common denominator for data interchange between different classes of software and applications.

```
......
Maximum Tempature F(C); Minimum Tempature F(C); <TD></TD>
82.0(27.8);62.1(16.7); In the <B>6 hours</B> preceding Oct 29,
1998 - 6:53 PM EST / 1998.10.29 2353 UTC
80.1(26.7);45.0(7.2);In the <B>24 hours</B> preceding Oct 28,
1998 - 11:53 PM EST / 1998.10.28 0453 UTC
......
```

Figure 6: A fragment of the comma-delimited file for the Savannah weather report page

By traversing the entire tree of the node `TABLE[2]` and applying the derived extraction rules, we may extract all the token values for each given token name in this region. Similarly, by traversing the entire tree of Savannah page, the semantic-token extractor produces as output a comma-delimited file for the Savannah weather report page. Figure 6 shows the portion of this comma-delimited file that is related to `TABLE[2]` node. The first line shows the name of the fields (the rows) that are being used. The second and third lines are two data records.

## 3.4. Hierarchical Structure Extraction

The goal of the hierarchical structure extractor is to make explicit the meaningful hierarchical structure of the original document by identifying which parts of the regions or token streams should be grouped together. More concretely, this step determines the nesting hierarchy (syntactic structure) of the source page, namely what kind of hierarchical structure the source page has, what are the top-level sections (tables) that forms the page, what are the sub-sections (or columns, rows) of a given section (or table), etc. The outcome of this step is the set of hierarchical structure extraction rules specified in a context-free grammar, describing the syntactic structure of the source document page. The following simple heuristics are frequently used by the hierarchy extractor to make the first guess of the sections and the nesting hierarchy of sections in the source document to establish the starting point for feedback-driven interaction with the user. These heuristics are based on the observation that the font size of the heading of a sub-section is generally smaller than that of its parent section.

- Identifying all regions that are siblings in the parse tree, and organizing them in the sequential order as they appear in the original document.
- Obtaining a section heading or a table name using the paired header tags such as `<H3>`, `</H3>`.
- Inferring the nesting hierarchy of sections or the columns of tables using font size and the nesting structure of the presentation layout tags, such as `<TR>`, `<TD>`, `<P>`, `<DL>`, `<DD>`, and so on.

We develop a hierarchical structure extraction algorithm that, given a page with all sections and headings identified, outputs a hierarchical structure extraction rule script expressed in an XML-compliant template for the page. Figure 7 shows a fragment of the XML template file corresponding to the part of a NWS weather report page shown in the right side of Figure 7. It defines the nesting hierarchy, annotated with some processing instructions.

The use of XML templates to specify the hierarchical structure extraction rule facilitates the code generation of the XWRAP for several reasons. First, XML templates are well-formed XML files that contain processing instructions. Such instructions are used to direct the XWRAP XML-template engine to the special placeholders where data fields should be inserted into the template. For instance, the processing instruction XG-InsertField-XG has the canonical form of `<?XG-InsertField-XG ''FieldName"?>`[3]. It looks for a field with a specified name "FieldName" in the comma-delimited file and inserts that data at the point of the processing instruction. Second, an XML template also contains a repetitive part, called XG-Iteration-XG, which is necessary for describing the nesting structure of regions and sections of a web page. The XG-Iteration-XG processing instruction determines the beginning and the end of a repetitive part. A repetition can be seen as a loop in classical programming languages. After the template engine reaches the "End" position in a repetition, it takes a new record from the delimited file and goes back to

[3] XG stands for **X**WRAP code **G**enerator.

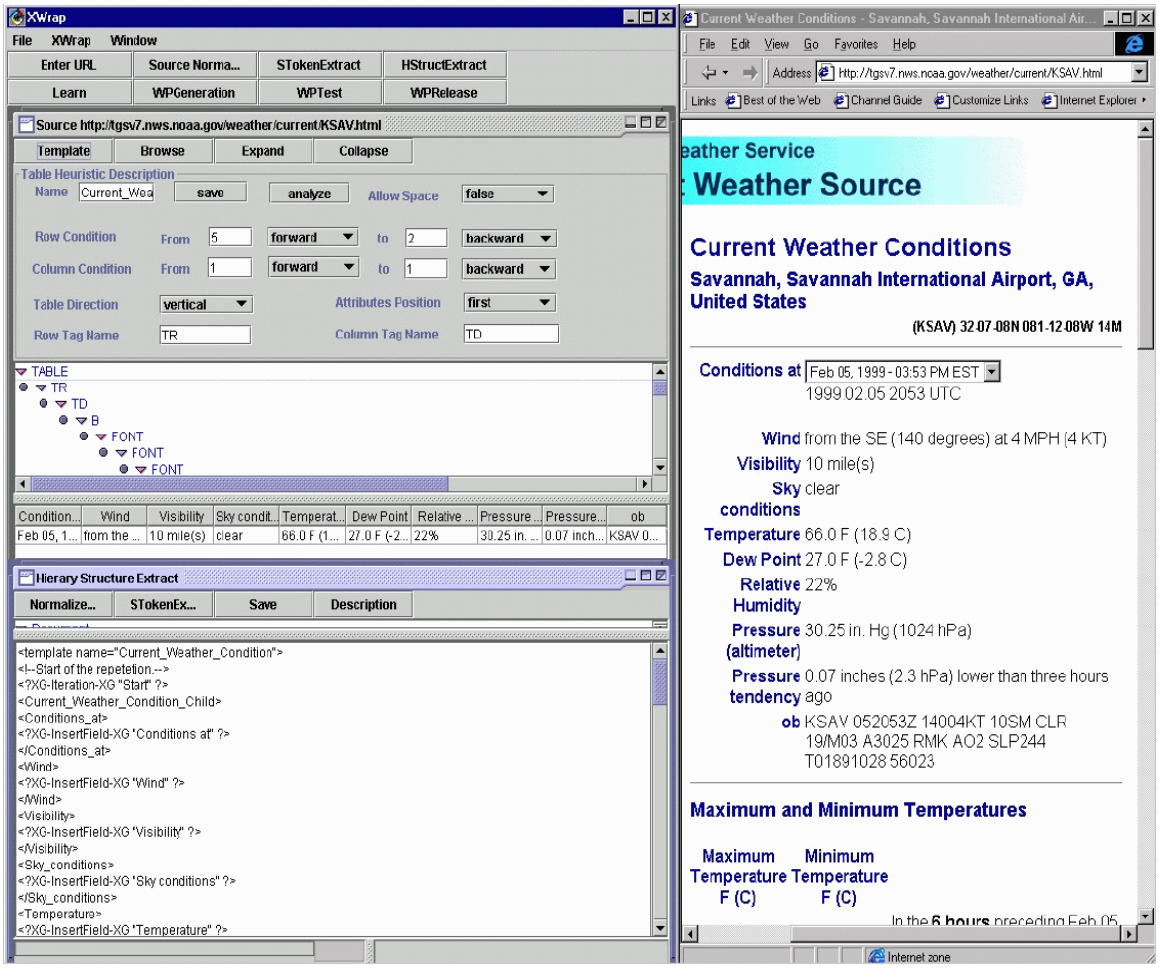


Figure 7: A screenshot of the Hierarchical Structure Extraction Window

the "Start" position to create the same set of XML tags as in the previous pass. New data is inserted into the resulting XML file.

The XWRAP code generator generates the wrapper code for a chosen web source by applying the comma-delimited file (as shown in Figure 6 for the running example), the region extraction rules (as given in Example 2), and the hierarchical structure extraction rules (see Figure 7), all described using the XWRAP's XML template-based extraction specification language. Due to the space limitation, the details on the language and the example code generated by XWRAP are omitted here. Readers may visit the XWRAP web site `www.cc.gatech.edu/projects/disl/XWRAP` for further details.

# 4. Experimental Results

## 4.1. Representative Web Sites

We have chosen 4 web sites that are representative in our opinion to report our experiments.

1. NOAA weather site shown in Figure 2 and Figures 7. NOAA pages combine multiple small tables (vertical or horizontal) with some running text. Number of random samples collected: 10 different pages.

2. Buy.com, a commercial web site [www2.buy.com] with many advertisements and long tables. This is a web site with frequent updates of content and changes of format. It is an example of challenging sites for wrapper generators. Web pages used

| Data Source | Generation Time(minutes) | Revision (times) | Extraction Rules Length(lines) | XML Template Length(lines) | Accuracy Verification |
|---|---|---|---|---|---|
| NOAA | 40 | 2 | 114 | 153 | 100% |
| CIA Factbook | 25 | 1 | 237 | 23 | 100% |
| Buy.com | 16 | 0 | 102 | 46 | 100% |
| Stockmaster | 23 | 1 | 90 | 46 | 100% |

Figure 8: XWRAP Performance Results

| Data Source | Avg. vs. St. Dev. | Document Size(byte) | Document Tree Length | Result XML Size(byte) | Doc/XML |
|---|---|---|---|---|---|
| NOAA | Average | 31135 | 1145 | 7593 | 4.1 |
| | St. Dev. | 465 | 23 | 42 | 0.1 |
| CIA Factbook | Average | 16115 | 834 | 18981 | 0.98 |
| | St. Dev. | 4503 | 188 | 5623 | 0.1 |
| Buy.com | Average | 44075 | 832 | 5172 | 9.6 |
| | St. Dev. | 11871 | 232 | 2014 | 3.4 |
| Stockmaster | Average | 21218 | 523 | 370 | 57.3 |
| | St. Dev. | 1137 | 32 | 11 | 2.4 |

Figure 9: Performance Statistics w.r.t. source document size and result XML size

in our evaluation are generated dynamically by a search engine. Pages used include book titles that contain keywords such as "JDBC" and "college life". Number of random samples: 20 pages.

3. Stockmaster.com, another commercial site with advertisements, graphs, and tables. This is an example of sites with extremely high frequency updates. Pages used in our evaluation are also generated dynamically, including stock information on companies such as IBM and Microsoft. Number of random samples: 21 pages.

4. CIA Fact Book, a well-known web site (www.odci.gov/cia/publications/factbook) used in several papers [16, 2]. Although infrequently updated, it is included here for comparison purposes. Number of random samples: 267 pages.

### 4.2. Evaluation of Wrapper Generation

The first part of experimental evaluation of XWRAP concerns the wrapper generation process. We measured the approximate time it takes for an expert wrapper programmer (in this case an experienced graduate student) to generate wrappers for the above 4 web sites. The results are shown in Figure 8.

Our initial experience tells us that the main bottleneck in the wrapper generation process is the number of iterations needed to achieve a significant coverage of the web site. The main advantage of our wrapper is the level of robustness. The wrappers generated by XWRAP can handle pages that have slightly different structure (such as extra or missing fields (bullets or sections) in a table (a text section) than the example pages used for generating the wrapper. However, when the pages are significantly different from the example pages used in the wrapper generation process, the wrapper will have to be refined. Several improvements on the GUI have been made since this experiment to further shorten the wrapper generation process.

### 4.3. Evaluation of Wrapper Execution

All measurements of wrapper executions were carried out on a dedicated 200MHz Pentium machine (jambi.cse.ogi.edu). The machine runs Windows NT 4.0 Server and there is only one user in the system. All the XWRAP software is written in Java. The main Java package used is Swing.

Figure 9 shows the first characterization of web page samples. We see that NOAA and Stockmaster.com have high uniformity (low standard deviation) in document size, due to their form-oriented page content (standard weather reports and standard stock price reports). The CIA Fact Book has medium standard deviation in document size, since the interesting facts vary somewhat from place to place. The Buy.com pages have high variance in document size, since the number of books available for each selection topic varies greatly. For variable-sized pages in Buy.com and CIA

| Data Source | Avg. vs. St. Dev. | Fetch Time(ms) | Expand Tree Times(ms) | Extraction Times(ms) | Generate Times(ms) | Total Time(ms) | Correlation Doc/Time |
|---|---|---|---|---|---|---|---|
| NOAA | Average | 4391 | 8531 | 3841 | 1128 | 18520 | 0.45 |
| | St. Dev. | 1032 | 1055 | 228 | 116 | 1636 | |
| CIA Factbook | Average | 1907 | 11916 | 4709 | 3902 | 23043 | 0.93 |
| | St. Dev. | 265 | 3366 | 1175 | 1297 | 5776 | |
| Buy.com | Average | 6908 | 7777 | 2748 | 838 | 18909 | 0.66 |
| | St. Dev. | 4333 | 1553 | 1439 | 287 | 6602 | |
| Stockmaster | Average | 1972 | 5489 | 1412 | 468 | 9973 | 0.35 |
| | St. Dev. | 489 | 453 | 497 | 121 | 1131 | |

Figure 10: Performance of Wrappers w.r.t. Fetch, Expand, Extract, and Result Generate time

Fact Book, we calculated the correlation between the input document size and the output XML file size (from the data table not shown in the paper due to space constraints). The correlation is strong: 1.00 for Buy.com and 0.98 for CIA Fact Book. This shows consistent performance of wrappers in mapping input to output.

Another interesting observation shown Figure 9 is the fact that the wrapper-generated document tree length is proportional to the input document size, and this, however, may not be true for the result XML file size. We call wrappers that ignore a significant portion of the source pages (in this case, the advertisements in Buy.com and Stockmaster.com) *low selectivity* wrappers. In our case, Buy.com and Stockmaster.com are low selectivity due to heavy advertisement, and their Input-Doc-Size/Output-XML-Size ratio is high (9.6 and 57.3, respectively). Purely informational sites such as NOAA and CIA Fact Book tend to have high selectivity (4.1 and 0.98, respectively).

Figure 10 shows the summary of execution (elapsed) time of wrappers. It is comforting that form-oriented pages (NOAA and Stockmaster.com) take roughly the same time (standard deviation at about 10% of total elapsed time) to process. This is the case for both a high selectivity site such as NOAA and a low selectivity site such as Stockmaster.com. For variable-sized pages in Buy.com and CIA Fact Book, we calculated the correlation between the input document size and total elapsed processing time: 0.66 for Buy.com and 0.93 for CIA Fact Book. The higher correlation of CIA Fact Book is attributed to its high selectivity (same input and output size), and lower correlation of Buy.com to its lower selectivity (input almost 10 times the output size). This shows the consistent performance of wrappers in elapsed time.

Figure 10 also shows that most of the execution time (more than 90%) is spent in four components of the wrapper: Fetch, Expand, Extract, and Generate. The first component, Fetch, includes the network access to bring the raw data and the initial parsing. Since we have no control over the network access time, the fetch time has high variance. This is confirmed by the lowest variance of the smallest documents (CIA Fact Book) and highest variance of largest documents (Buy.com).

The second component, Expand, consumes the largest portion of execution time. It is a utility routine that invokes Swing to expand a tree data structure for extraction. This appears to be the current bottleneck due to the visualization oriented implementation of Swing, and it is a candidate for optimization.

The third component, Extract, also uses the Swing data structure to do the Information Extraction phase (Section ). This phase does more useful work than Expand, but it is also a candidate for performance tuning when we start the optimization of the Expand component.

The fourth component, Generate, produces the output XML file. It is clearly correlated to the size of the output XML file. Except for the extremely short results from Stockmaster.com (consistently at about 370 bytes), the execution time of Generate for the other three sources is between 5 and 6 bytes of XML generated per 1 ms.

## 5. Conclusion

We have presented the XWRAP approach to semi-automatically generating wrappers for Web information sources and reported our initial experiments on performance of the XWRAP code generation system and the wrapper programs generated by XWRAP. Our wrapper generation framework has three distinct features. First, it explicitly separates tasks of building wrappers that are specific to a Web source from the tasks that are repetitive for any source, and uses a component library to provide basic building blocks for

wrapper programs. Second, it provides a user-friendly interface program to allow wrapper developers to generate their wrapper code with a few mouse clicks. Third and most importantly, we introduce and develop a two-phase code generation framework. The first phase utilizes an interactive interface facility to encode the source-specific metadata knowledge identified by individual wrapper developers as declarative information extraction rules. The second phase combines the information extraction rules generated at the first phase with the XWRAP component library to construct an executable wrapper program for the given web source.

Our work continues along three dimensions. The first aspect focuses on providing better tools to incorporate various machine learning algorithms to enhance the robustness of information extraction rules. The second aspect is to enrich the XWRAP information extraction rule language and the component library with enhanced pattern discovery capability and various optimization considerations. The third aspect concerns the incorporation of Microsoft repository technology [4, 5, 6] to handle and manage the versioning issue and the metadata of the XWRAP wrappers. Furthermore, we are interested in investigating issues such as whether the ability of following hyperlinks should be a wrapper functionality at the level of information extraction or a mediator functionality at the level of information integration.

**Acknowledgement.** This research is partially supported by DARPA grant MDA972-97-1-0016 and a grant from Intel. We thank both past and present members of the XWRAP team, especially David Buttler for his contribution to the XWRAP toolkit, and Wei Tang for his integration with the Continual Queries project.

# References

[1] B. Adelberg. Nodose - a tool for semi-automatically extracting structured and semi-structured data from text documents. *ACM SIGMOD*, 1998.

[2] N. Ashish and C. A. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of Coopis Conference*, 1997.

[3] P. Atzeni and G. Mecca. Cut and paste. *Proceedings of 16th ACM SIGMOD Symposion on Principles of Database Systems*, 1997.

[4] T. Bergstraesser, P. Bernstein, S. Pal, and D. Shutt. Versions and workspaces in microsoft repositorys. *ACM SIGMOD*, 1999.

[5] P. Bernstein. Microsoft repository. *VLDB'97 Tutorial and ACM SIGMOD'96 Tutorial*, 1997.

[6] P. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt. Microsoft repository version 2 and the open information model. *Information Systems 24(2)*, 1999.

[7] H. Garcia-Molina and et al. The TSIMMIS approach to mediation: data models and languages (extended abstract). In *NGITS*, 1995.

[8] J. Hammer, M. Brennig, H. Garcia-Molina, S. Nesterov, V. Vassalos, and R. Yerneni. Template-based wrappers in the tsimmis system. In *Proceedings of ACM SIGMOD Conference*, 1997.

[9] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semi-structured data from the web. *Proceedings of Workshop on Management of Semi-structured Data*, pages 18–25, 1997.

[10] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proceedings of AAAI Conference*, 1998.

[11] N. Kushmerick. Wrapper induction for information extraction. In *Ph.D. Dissertation, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04*, 1997.

[12] N. Kushmerick, D. Weil, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of Int. Joint Conference on Artifical Intelligence (IJCAI)*, 1997.

[13] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[14] L. Liu, C. Pu, W. Tang, J. Biggs, D. Buttler, W. Han, P. Benninghoff, and Fenghua. CQ: A Personalized Update Monitoring Toolkit. In *Proceedings of ACM SIGMOD Conference*, 1998.

[15] D. Raggett. Clean Up Your Web Pahes with HTML TIDY. *http://www.w3.org/People/Raggett/tidy/*, 1999.

[16] A. Sahuguet and F. Azavant. WysiWyg Web Wrapper Factory (W4F). *Proceedings of WWW Conference*, 1999.

[17] S. Soderland. Learning to extract text-based information from the world wide web. *Proceedings of Knowledge Discovery and Data Mining*, 1997.

[18] W3C. Reformulating HTML in XML. *http://www.w3.org/TR/WD-html-in-xml/*, 1999.