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

Ling Liu, Calton Pu, Wei Han

Oregon Graduate Institute of Science and Technology
{lingliu,calton,weihan}@cse.ogi.edu

Abstract

The amount of useful semi-structured data on the web continues to grow at a stunning pace. Often interesting web data are not in database systems but in HTML pages, XML pages, or text files. Data in these formats is not directly usable by standard SQL-like query processing engines that support sophisticated querying and reporting beyond keyword-based retrieval. Hence, the web users or applications need a smart way of extracting data from these web sources. One of the popular approaches is to write wrappers around the sources, either manually or with software assistance, to bring the web data within the reach of more sophisticated query tools and general mediator-based information integration systems.

In this paper, we describe the methodology and the software development of an XML-enabled wrapper construction system - XWRAP 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 inductive learning algorithms that derive or discover wrapper patterns by reasoning about sample pages or sample specifications. 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. The two-phase code generation approach exhibits 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 mouse 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. We report the performance of XWRAP and our experiments by demonstrating the benefit of building wrappers for a number of Web sources in different domains using the XWRAP generation system.

*This research is partially supported by DARPA contract MDA972-97-1-0016 and a grant from Intel.

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. This has been so successful that many Net businesses rely on advertisement as their main source of income, offering free email services, for example. The bad news is that these "human-oriented" HTML pages are difficult for programs to parse and capture. Furthermore, the rapid evolution of Web pages requires making corresponding changes in the programs accessing them. In addition, most of the web information sources are created and maintained autonomously, and each offers services independently. Interoperability of the web information sources remains the next big challenge.

A popular approach to address these problems is to write *wrappers* to encapsulate the access to sources. For instance, the most recent generation of information mediator systems (e.g., Ariadne [20], CQ [24, 25], Internet Softbots [23], TSIMMIS [14, 16]) 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 a systematic 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 doucment, to distinguish the information content from their metadata description, and to recognize and encode the metadata explicitly in the XML output. Our main contribution here is to provide a set of interactive mechanisms and heuristics for generating information extraction rules with a few clicks, and a way to combine those information extraction rules into a method for generating an executable wrapper program.

This is not the first time the problem of information extraction from a Web document has been addressed. [6, 17] 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. [2, 28, 5, 20, 13, 22, 23, 29] 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.

Our approach differs from these proposals in two distinct ways. First, we introduce a two-phase code generation approach for wrapper generation. 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:

1. it provides a user-friendly interface program to allow users to generate their information extraction rules with a few clicks.
2. 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.
3. it facilitates the use of the micro-feedback approach to revisit and tune the wrapper programs at run time.

Second, we divide the task of identifying object boundaries into two steps: region identification and semantic token identification (see Section 4). Once a Web document is fetched, XWRAP build a parse tree with HTML tags

as internal nodes and information content as leaf nodes. The structure of the tree follows the nested structure of start- and end-tags. Users may highlight a specific word or phrase or sentence as the starting point of a meaningful region. XWRAP will then apply the heuristics on nearest region tags to derive the type of the region. Then the heuristics for identifying features of a specific region are applied. Similarly, users may identify semantic tokens of interest with a few clicks and fire learning algorithms to detect repetitive token patterns within a region. Finally, we provide a way to combine the region extraction rules and semantic token extraction rules generated to determine the hierarchical structure of the regions or semantic tokens of interest. We applied the XWRAP approach to four different application areas using Web documents obtained from ten different sites, which together contained thousands of objects (section 5). The results were uniformly good, gaining 100% accuracy in all sites examined (see Section 5). Furthermore, we want to leverage on standards as much as possible, thus choosing XML as our output format. The development of XWRAP presents not only a software tool but also the methodology for developing an XML-enabled, feedback-based, interactive wrapper construction facility that generates value-added wrappers for Internet information sources.

Before explaining the details of our approach, we would like to note that semi-automated wrapper construction is just one of the challenges in building a scalable and reliable mediator-based information integration system for Web information sources. The other important problems include resolving semantic heterogeneity among different information sources, efficient query planning for gathering and integrating the requested information from different Web sites, and intelligent caching of retrieved data, to name a few. The focus of this paper is solely on wrapper construction.

The rest of the paper proceeds as follows. We overview the methodology for semi-automatic wrapper construction in Section 2. We describe the XWRAP technology for information extraction and for constructing wrappers for web information sources in Section 3 and Section 4. We demonstrate the effectiveness of our wrapper construction techniques through an analysis of our experimental results in Secton 5. We conclude the paper with a discussion on related work in Section 6 and a summary and an outline of future directions in Section 7.

2 The Design Framework: An Overview

2.1 Architecture

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. 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 ruls produced in the second step. A key 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 4.3). The code generator interpret the XML-template rules by linking each

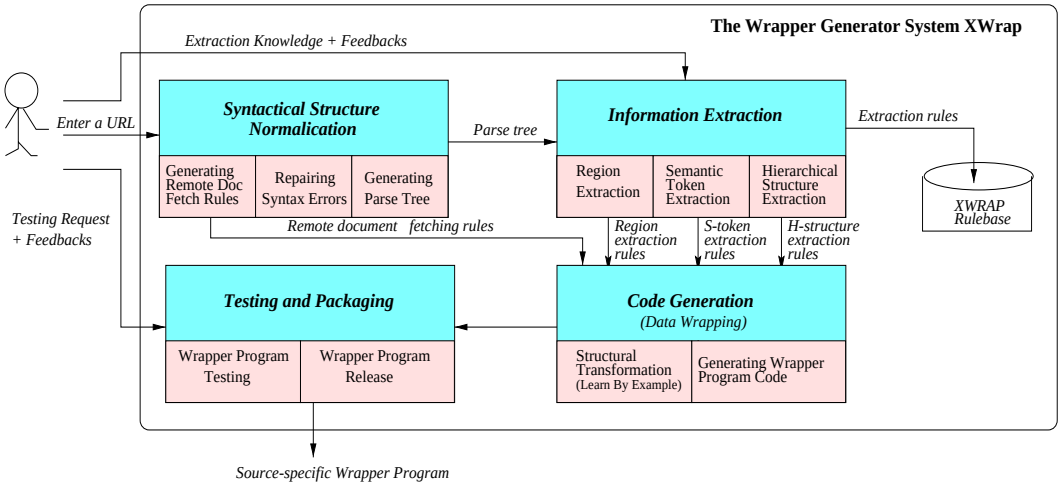

Figure 1: XWRAP system architecture for data wrapping

executable components 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 2) 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.

The XWRAP architecture for data wrapping is motivated by the design decision fortaking advantage of declarative language for specification of information extraction knowledge, for exploring reusable functionality, and for separating data wrapping from functional wrapping.

2.2 Phases and Their Interactions

As the wrapper-generation process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the construction process as occurring in one single step. For this reason, we partition the wrapper construction process into a series of subprocesses called *phases*, as shown in Figure 3. A phase is a logically cohesive operation that takes as input one representation of the source document and produces as output another representation.

XWRAP goes through six phases to construct and release a wrapper. Tasks within a phase run concurrently using a synchronized queue; each runs its own thread. For example, we decide to run the task of fetching a remote document and the task of repairing the bad formatting of the fetched document using two concurrently synchronous threads in a single pass of the source document. The task of generating a syntactic-token parse tree from an HTML document requires as input the entire document; thus, it cannot be done in the same pass as the remote document fetching and the syntax reparation. Similar analysis applies to the other tasks such as code

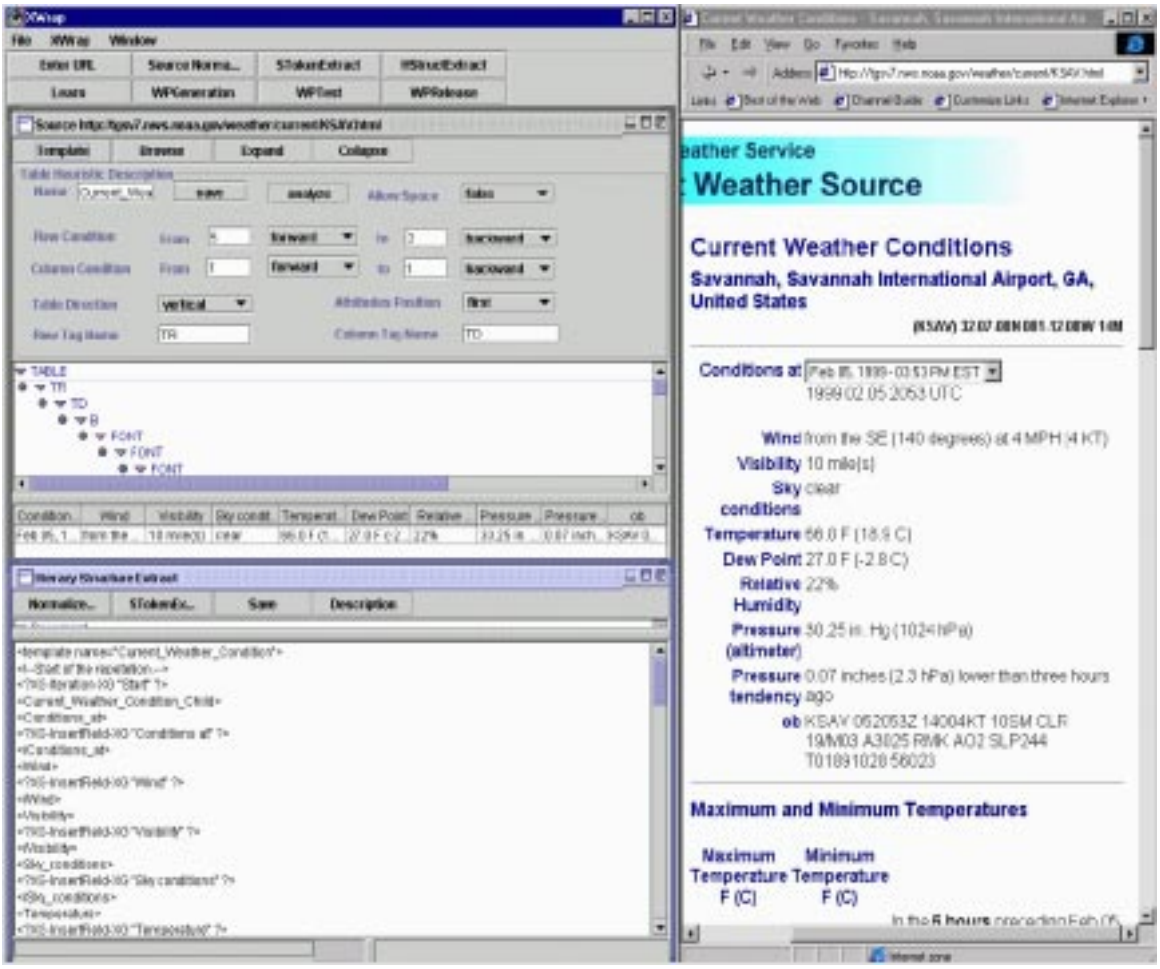

Figure 2: A screenshot of the Hierarchical Structure Extraction Window

generation, testing, and packaging.

The interaction and information exchange between any two of the phases is performed through communication with the bookkeeping and the error handling routines. The *bookkeeping* routine of the wrapper generator collects information about all the data objects that appear in the retrieved source document, keeps track of the names used by the program, and records essential information about each. For example, a wrapper needs to know how many arguments a tag expects, whether an element represents a string or an integer. The data structure used to record this information is called a symbol table.

The *error handler* is designed for the detection and reporting errors in the fetched source document. The error messages should allow the wrapper developer to determine exactly where the errors have occurred. Errors can be encountered at virtually all the phases of a wrapper. Whenever a phase of the wrapper discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the wrapper must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors. Good error handling is difficult because certain errors can mask subsequent errors. Other errors, if not properly handled, can spawn an avalanche of spurious errors. Techniques for error recovery are beyond the scope of this paper.

In the subsequent sections, we focus our discussion primarily on information extraction component of the XWRAP,

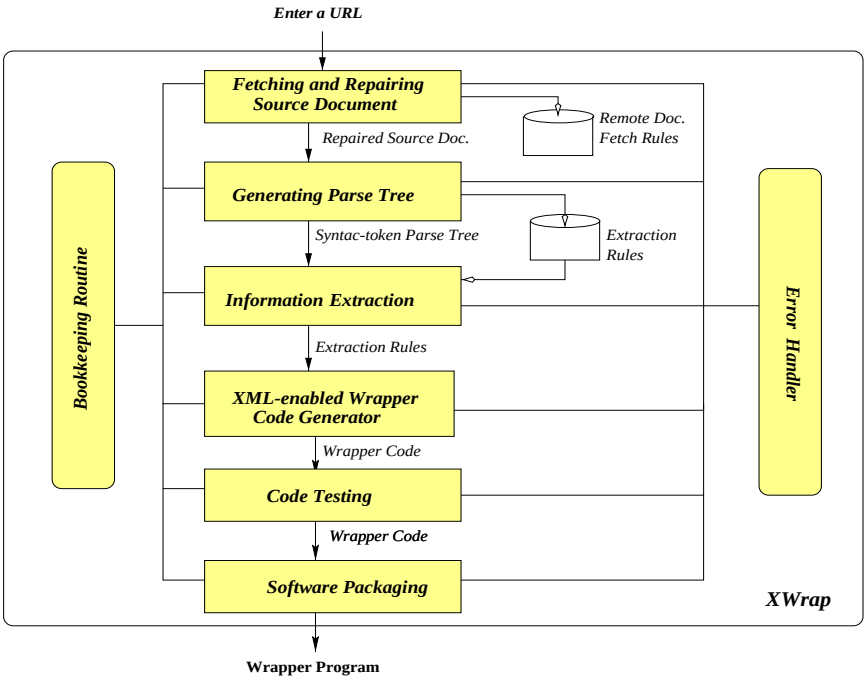

Figure 3: Data wrapping phases and their interactions

and provide a walkthrough example to illustrate how the three sets of information extraction rules are identified, captured, and specified. As the syntactical structure normalization is a necessary preprocessing step for information extraction, a brief description of the syntactic normalizer is also presented.

3 Preprocessing: Syntactical Structure Normalization

The Syntactical Structure Normalization process is carried out in two phases, as shown in Figure 3. The first phase consists of two concurrently synchronous tasks - remote document retrieval and syntax reparation. The second phase is responsible for generating a syntactic-token parse tree, of the repaired source document.

3.1 Fetching a Web Page

The Remote Document Retrieval component is responsible for generating a set of rules that describe the list of interface functions and parameters as well as how they are used to fetch a remote document from a given web source. The list of interface functions include the declaration to the standard library routines for establishing the network connection, issuing an HTTP request to the remote web server through a `HTTP Get` or `HTTP Post` method, and fetching the corresponding web page. Other desirable functions include building the correct URL to access the given service and pass the correct parameters, and handling redirection, failures, or authorization if necessary.

For each wrapper, there is a set of retrieval rules. Each rule specifies the name of the rule, the list of parameters it takes, the built-in functions `GetURL` or `PostURL`, the type of the URL protocols like *http, file, and ftp*, the protocol-specific remote fetch method (such as `HTTP GET` and `HTTP POST`), and the corresponding URL. XWRAP will automatically take care of packing the URL request parameters in the correct way as required by the `HTTP GET` and `HTTP POST`) protocol variants. In the case of an *PostURL* request, the correct construction of the parameter

object needs to be deduced from the web form where the URL request originates. The HTTP specification requires that the POST parameters be submitted in the order they appear in the form of the page.

Assume we want to construct a wrapper for noaa current weather report web site, and the URL entered at the start of XWRAP is http://weather.noaa.gov/cgi-bin/currwx.pl?cccc=KSAV, asking for the current weather in Savannah. Figure 4 shows a remote document retrieval rule derived from the given URL. It uses the XWRAP library function URLGet(...). The regular expression specified by Match(K[A-Z]{3}) specifies that the location code is restricted to four capital alphabet characters, starting with the character "K". When a web site offers more than one type of search capability, more than one retrieval rules may need to be generated.

```
Remote_Document_Fetch_Rules(XWRAP-weather.noaa.gov)::
   GetURL(String location-code)
   {
        Protocol:  HTTP;
        Method:  GET;
        URL: http://weather.noaa.gov/cgi-bin/currwx.pl?cccc=?location-code;
        ParaPattern:  location-code, match(K[A-Z]{3});
   }
```

Figure 4: Example rules for fetching remote documents

3.2 Repairing Bad Syntax

As soon as the first block of the source document is being fetched over, the syntax repairing thread begins. It runs concurrently with the Remote Document Retrieval thread, and repairs bad HTML syntax. This step inserts missing tags, removes useless tags, such as a tag that either starts with ¡! Pr is an end tag that has no corresponding start-tag. It also repairs end tags in the wrong order or illegal nesting of elements. We describe each type of HTML errors in a normalization rule. The same set of normalization rules can be applied to all HTML documents. Our HTML syntax error reparation module can clean up most of the errors listed in HTML TIDY [27, 30].

3.3 Generating a Syntactic Token Tree

Once the HTML errors and bad formatting are repaired, 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>. Different languages may define which is called a token differently. For HTML pages, the usual tokens are paired HTML tags (e.g., <TR>, </TR>), singular HTML tags (e.g.,
, <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 5). and a fragment of HTML document for this paper in Figure 6.

Figure 7 shows a portion of the HTML tree structure, corresponding to the above HTML fragment, which is generated by running a 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

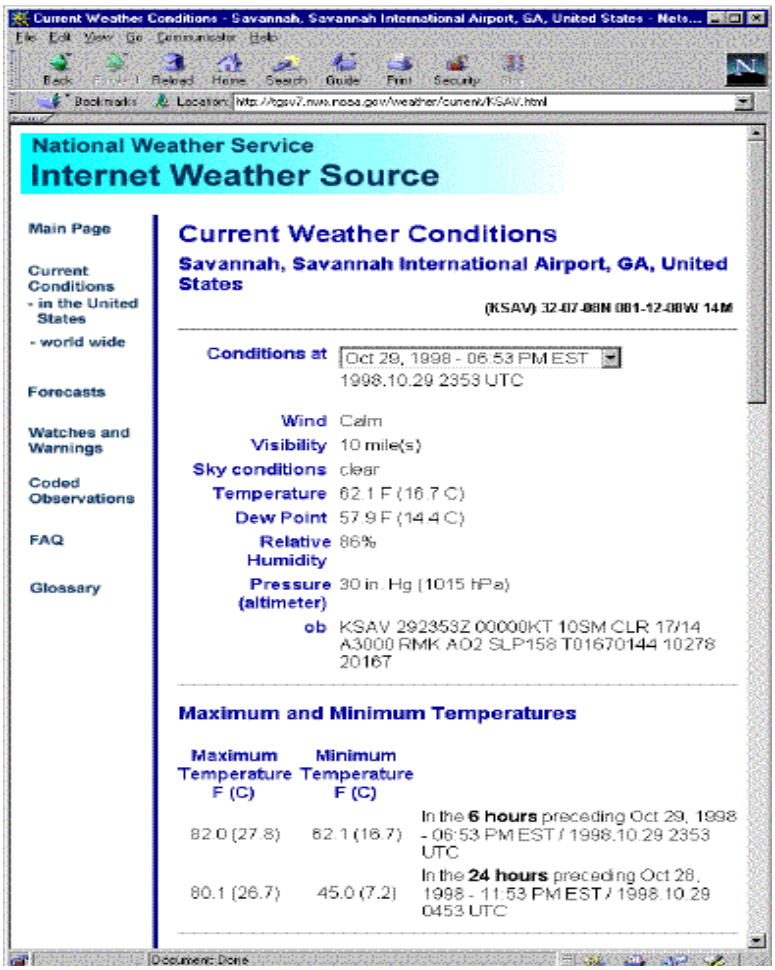

Figure 5: An example weather report page at the nws.noaa.gov site

token nodes at leaf node level, such as `Maximum Tempature`, `Minimum Tempature`, `84.9(29.4)`, `64.0(17.8)`, etc.

Important to note is that every syntactic token parse tree is organized as follows. All non-leaf nodes are tags and all leaf nodes are text strings, each in between a pair of tags. XWRAP defines a set of tree node manipulation functions for each tree node object, including `getNodeType(node_id)`, `getNodeName(node_id)`, `getNodeId(String NN)`, and `getNodePath(node_id)`, in order to obtain the node type - tag node or leaf (value) node, the node name - tag name or text string, the node identifier for a given string, or the path expression from the root to the given node. 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.

4 The Methodology for 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

```
<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 6: An HTML fragment of the weather report page at nws.noaa.gov site

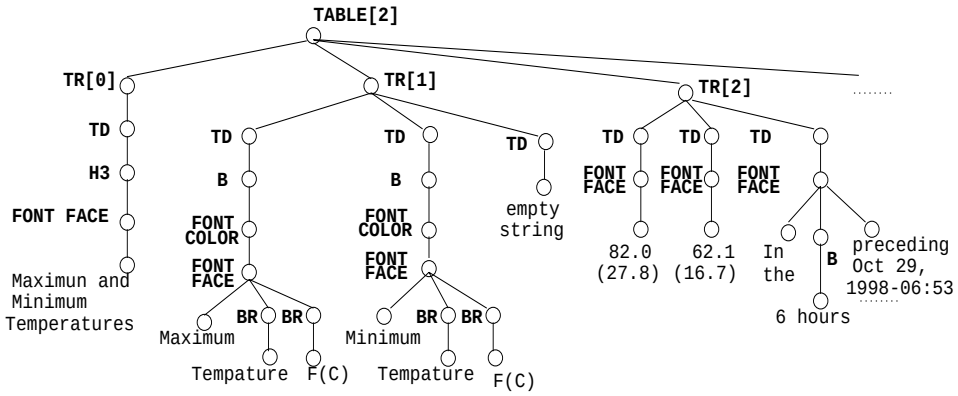

Figure 7: A fragment of the HTML tree for the Savannah weather report page

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; each step generates a set of extractions rules to be used by the code generation phase to generate wrapper program code.

- Step 1: *Identifying regions of interest on a page*
  This step 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 can identify regions of interest from the parse tree.
- Step 2: *Identifying semantic tokens of interest on a page.*
  This step is carried out by an interactive program, called *semantic-token extractor*, which allows 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 is the set of semantic token extraction rules that can locate and extract the semantic tokens of interest, and a comma-delimited file containing all the element type and element value pairs of interest.
- Step 3: *Determining the nesting hierarchy for the content presentation of a page.*

This step is performed by the hierarchical structure extractor, which infers and specifies the nesting structure of the sections of a web page (document) being wrapped. Such hierarchical specification will be used for content-sensitive information extraction from the source document(s). 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.

Important to note is that, for structured data sources such as database sources or XML documents, the information extraction process can be conducted automatically, following the table schema or the XML tags. However, this is not the case for unstructured or semi-structured information sources such as HTML documents or text files, because semi-structured or unstructured data is provided with no self-describing properties. Therefore, our goal is to perform the information extraction with minimal user interaction.

In summary, the semantic token extractor analyses the parse tree structure of the source document and its formatting information, and guesses the semantic tokens of interest on that page based on a set of token-recognition heuristics. Similarly, the hierarchical structure extractor also uses the formatting information and the source-specific structural rules to hypothesize the nesting structure of the page. The heuristics used for identifying important regions and semantic tokens in a page and the algorithms used to organize interesting regions of the source page into a nested hierarchy are an important contribution of this work. We describe them in more detail below.

4.1 Region Extraction: Identifying Important Regions

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 8 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 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 5), and a fragment of HTML parse tree as shown in Figure 7). To identify and locate the region of the table node TABLE[2], we apply the region extraction rules given in Figure 8 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`. With these index information, the region

```
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 8: Extraction rules for a table region in an HTML page

extractor can easily identify the effective table region, the area that does not include the row for table name and the empty row.

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, we need the user to highlight the table name node in the parse tree window (recall Figure 2). Based on the user's input, XWRAP can infer the path expression for the table name node is
   `TABLE[2].TR[0].TD[0].H3[0].FONT[0].FONT[0]`.
   Then by applying the fifth region extraction rule `getTableInfo`, we can extract the table name. Note that the function `getTableName(node_id)` calls the the following semantic token extraction rule to obtain the actual string of the table name (see Section 4.2 for details on semantic token extraction).

```
<ST_extract> ST_extract(String TN)
   <rule_exp>
     extract TN = TABLE[2].TR[0].TD[0].getChildNode(1).getStoken()
     where   TABLE[2].TR[0].TD[0].getChildNode(1).getNodeType() = 'tag'
     and     TABLE[2].TR[0].TD[0].getChildNode(1).getNodeName() = 'H3';
   </rule_exp>
</ST_extract>
```

The path of this table name node can be computed directly by invoking
`getNodePath(getNodeId('TABLE[2]'))`, which returns

`HTML.BODY.table[0].TR[0].TD[4].TABLE[2].TR[0].TD[0].H3[0].FONT[0].FONT[0]`.

It is important to note that the design of our region extraction rules is robust in the sense that the extraction rules are carefully designed to compute the important information (such as the number of tables in a page, the number of attributes in a table, etc.) at runtime. For example, let us assume that the nws.noaa.gov wrapper was constructed using the example page from a Portland weather report at a specific time, which happens to contain only three tables instead of the normal layout of four tables. The first table contains only 7 rows instead of the normal layout of 9 rows. When the very same wrapper runs to extract the page of Savannah, GA, our wrapper will automatically deduce that the page has four tables and the first table has 9 rows, rather than assuming all the weather report at nws.noaa.gov obey the same format. Furthermore, our region extraction rules are defined in a declarative language and therefore independent of the implementation of the wrapper code. This higher level of abstraction allows the XWRAP wrappers to enjoy better extensibility and ease in maintenance and in adapting to changes at the source.

4.2 Semantic Token Extraction: Finding Important Semantic Tokens

In general each semantic token is a sub-string of the source document that is to be treated as a single logical unit. There are two kinds of token: specific strings such as HTML tags (e.g., `TABLE`, `FONT`), and semantic tokens such as those strings in between a pair of HTML tags. To handle both cases, we shall treat a token as a pair consisting of two parts: a token name and a token value. For a tag token such as `FONT`, the tag name is `FONT` and the tag value is the string between a beginning tag `<FONT>` and its closing tag `</FONT>`. A semantic token such as `Maximum and Minimum Tempature F(C)` or `Current Weather Conditions` in between the start and end tags of the tag token `FONT` will be treated as either a name token or a value token, depending on the context or the user's choice. Similar treatment applies to the token such as `Savannah, Savannah International Airport, GA, United States`. In addition, a help function - `getStoken(node_id)` is provided for semantic token extraction rules. It extracts and concatnates all text strings from the leaf nodes of the subtree identified by the given `node_id`.

The main task of a semantic token extractor (S-token extractor for short) is to find semantic tokens of interest, define extraction rules to locate such tokens, and specify such tokens in a comma-delimited format[1], which will be used as input in the code generation phase. 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 7. 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 Tempature 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[][])
```

[1] 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.

```
<!-- 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>
```

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 9 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.

```
......
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 9: A fragment of the comma-delimited file for the Savannah weather report page

4.3 Hierarchical Structure Extractor: Obtaining the Nesting Hierarchy of the Page

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.

Similar to the semantic token extractor, the hierarchical structure can be extracted in a semi-automatic fashion for a larger number of pages. By semi-automatic we mean that the task of identifying all sections and their nesting hierarchy is accomplished through minimal interaction with the user. The following simple heuristics are most 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 10 shows the fragment of the XML template file corresponding to the part of a NWS weather report page shown in Figure 7. It defines the nesting hierarchy, annotated with some processing instructions.

```
.....
<Maximum_and_Minumum_Temperatures>
<Description>Maximum and Minimum Temperatures</Description>
<!-- Start of the repetition -->
<?XG-Iteration-XG ``Start"?>
  <Maximum_and_Minimum_Temperatures_Child>
     <Maximum_Temperature>
     <Description>MaximumTemperature F(C)</Description>
     <Value><?XG-InsertField-XG ``Maximum Tempature"></Value>
     </Maximum_Temperature>

     <Minimum_Temperature>
     <Description>MinimumTemperature F(C)</Description>
     <Value><?XG-InsertField-XG ``Minimum Temperature"></Value>
     </Minimum_Temperature>

     <TD>
     <Description></Description>
     <Value><?XG-InsertField-XG ``TD"></Value>
     </TD
  </Maximum_and_Minumum_Temperatures_Child>
<?XG-Iteration-XG ``End"?>
<!-- End of the repetition -->
</Maximum_and_Minumum_Temperatures>
......
```

Figure 10: A fragment of the hierarchical structure extraction rule for nws.noaa.gov current weather report page

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 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 is <?XG-InsertField-XG ``FieldName"?>. 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 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.

Due to the fact that the heuristics used for identifying sections and headings may have exceptions for some information sources, it is possible for the system to make mistakes when trying to identify the hierarchical structure of a new page. For example, based on the heuristic on font size, the system may identify some words or

phases as headings when they are not, or fail to identify phases that are headings, but do not conform to any of the pre-defined regular expressions. We have provided a facility for the user to interactively correct the system's guesses. Through a graphical interface the user can highlight tokens that the system misses, or delete tokens that the system chooses erroneously. Similarly, the user can correct errors in the system generated grammar that describes the structure of the page.

The XWRAP code generator generates the wrapper code for a chosen web source by applying the comma-delimited file (as shown in Figure 9 for the running example), the region extraction rules (as given in Example 2), and the hierarchical structure extraction rules (see Figure 10), all described using the XWRAP's XML template-based extraction specification language. Due to the space limitation, the details on the language is omitted here.

Finally, to satisfy the curiosity of some readers, we show in Appendix A a fragment of the XML document transformed from the original HTML page by the XWRAP_nws.noaa.gov wrapper program, which was generated semi-automatically using XWRAP toolkit for the NWS web source.

5 Experimental Results

5.1 Representative Web Sites

Due to the rapid evolution of the web, there are few agreed upon standards with respect to the evaluation of web pages. Existing standard benchmarks such as the SPECweb96, Webstone 2.X, and TPC-W impose a standard workload to measure server performance. Although it is an interesting challenge to collect a representative set of web sites for comparing the performance of web data source wrappers, that task is beyond the scope of this paper. For our analysis, we have chosen 4 web sites that are representative in our opinion:

1. NOAA weather site shown in Figures 2 and Figure 5. 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 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 [www.stockmaster.com] 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 (http://www.odci.gov/cia/publications/factbook), a well-known web site used in several papers [28, 5]. Although infrequently updated, it is included here for comparison purposes. Number of random samples: 267 pages.

5.2 Evaluation of Wrapper Generation

The first part of experimental evaluation of XWRAP concerns the wrapper generation process. Since the use of wrapper generator depends on many factors outside of our control, we avoid making any scientific claims of this evaluation result. The experiments are included so readers may gain an intuitive feeling of the wrapper generator usage.

We measured the approximate time it takes for an expert wrapper programmer (in this case a graduate student) to generate wrappers for the above 4 web sites. Since production-use wrappers are typically written and maintained by experienced professional programmers, this is a common case. The results are shown in Figure 11. We already have several improvements on the GUI that should shorten the wrapper generation process.

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 11: XWRAP Performance Results

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.

Our experience also tells us that the higher quality of the sample pages used for generating wrappers, the higher accuracy one would get. Since an XWRAP wrapper is generated "by example", the choice of a simplistic example page would produce too simple a wrapper for more complex pages. Typically, as more complex pages are encountered, the wrapper is refined to handle the new situation. Ideally, one would find the most complex example web page of the site, and use it to generate the "nearly complete" wrapper for that site. Developing mechanisms for selecting high quality sample Web pages is a topic of our ongoing research.

5.3 Evaluation of Wrapper Execution

Our current implementation has been built for extensibility and ease of software maintenance. Consequently, we have chosen software components with high functionality and postponed the optimization of data structures and algorithms to a later stage. One example of such trade-off is the use of Java Swing Class library to manage all important data structures such as the document tree. This choice minimizes the work for visualization of these data structures, which is more important than raw performance at this stage of XWRAP development.

All measurements 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.

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	0.9
	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 12: Performance Statistics w.r.t. source document size and result XML size

Figure 12 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.

Also from Figure 12 we see that the wrapper-generated document tree length is proportional to the input document size. However, this 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 0.9, respectively).

An expected, but important observation is about consistent performance of the wrappers, in terms of successfully capturing the information from source pages. First, form-oriented input pages such as NOAA and Stockmaster.com have high uniformity (low standard deviation) in the result XML file size. Second, for variable-sized pages in Buy.com and CIA 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.

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

Figure 13 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 13 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 4). 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.

6 Related Work

Recently considerable attention has been received on generating wrappers for web information sources and providing database like queries over semi-structured data through wrappers. We below summarize some of the popular projects and compare them with our XWRAP system.

TSIMMIS [16] developed a logical template-based approach to generating wrappers for web sources and other types of legacy systems. This approach provides a way of rapidly constructing wrappers by example but it could require a large number of examples to specify a single source.

The Internet robot project at University of Washington [12] developed an Internet comparison shopping agent that can automatically build wrappers for web sites. Since the proposed approach focuses more on pages that contain items for sale, much stronger assumptions are made about the type of information to be used to guess the underlying structure. As a result, their wrapper language is not very expressive, and the system is quite limited in terms of the types of pages for which it can generate wrappers.

Another endeavor on wrapper construction at University of Washington is made by Kushmerick et al. [22, 23] using inductive learning techniques. The proposed approach builds a program that extracts information from a web page based on a set of pre-defined extractors. The advantage of this approach is that the resulting wrappers will be more robust to inconsistencies across multiple-document pages. However, their approach could not be used to generate wrappers for more complex pages such as the NWS weather report pages, without first building extractors for each of the fields of those pages.

The wrapper construction effort in the ARIADNE project [5, 20] has also demonstrated the importance and feasibility of building a wrapper generator. The focus of their work is very similar to ours, i.e., semi-automatic generation of wrappers for the web sources to be integrated by a mediator or a software agent. However, they follow a very different approach that uses LEX to find tokens of interest from a source page and uses YACC to define and extract the nesting structure of the page. However, the current version of the ARIADNE system does not handle the web pages that contain tables such as the NWS weather report site. Also it does not provide feedback-based learning capability to enhance the robustness of generated wrappers in handling inconsistencies across multiple-instance pages.

A recent project W4F [28] at University of Pennsylvania produces a toolkit to help wrapper developers to develop wrappers. The main feature of W4F includes the use of the DOM object model instead of the grammar-based approach as in JEDI [18] and the use of the Nested String Language (NSL) to encode the information extraction rules.

Despite the commonality with other approaches such as encoding extraction rules in description files and using heuristics based on particular HTML tags, our wrapper generation approach differs markedly from the existing approaches. The most distinct feature is its unique two-phase code generation framework. 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. In addition, XWRAP 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.

In addition, a variety of research has been devoted to issues in directly querying semi-structured data from web sources in a database-like fashion [1, 6, 10, 21, 26]. These efforts are concerned with issues such as the development of data models and query languages for semi-structured data, defining formal semantics for such query languages, and efficiently implementing these languages.

Other interesting work includes WebL [19] and WIDL [3], offering some advanced features for the web document retrieval, and W4F [28], offering an interesting web wrapper factory to extract information using nested string lists as the target structure. In addition, Web-OQL [4] and XML-QL [11] provide queries with variable binding, and offer interesting techniques for implementing functional wrappers on top of the XWRAP data wrapper. Other interesting effort in using declarative approaches to information extraction includes QEL [15] and XML-Pointer [31], although their current developments are limited to simple constructs of the web pages.

There are some commercial wrapping services available on the Internet, such as Junglee (bought by Amazon.com), Jango (bought by Excite), and mySimon. They are able to extract information from large HTML sources. However, their technology is considered a business asset and proprietary, consequently unavailable to the world at large.

7 Conclusion

We have presented our approach for semi-automatically generating wrappers for Web information sources. There are three main contributions of the paper. First, we develop a two-phase code generation methodology and a set of mechanisms for semi-automatic construction of XML-enabled wrappers. Second, we explicitly separate tasks of building wrappers that are specific to a Web source from the tasks that are repetitive for any source, and provide a component library to host basic building blocks of wrapper programs. Third, XWRAP provides inductive learning algorithms that derive or discover wrapper patterns by reasoning about sample pages or sample specifications. The ideas and results of the XWRAP system appear to be effective for many semi-structured web sources. However, we need more advanced wrappers to be able to broaden the scope of sources we can generate wrappers for. Currently we are working on enhancing the generation of data wrappers with the capability of handling complex tables that have more than two-dimension and finer grained queries that have complex search conditions. We are also interested in mechanisms for enhancing the reliability of the wrappers generated.

Our future work will involve three distinct aspects of the system. The first aspect focuses on providing better tools to assist user in choosing sample Web pages from the given Web site and to incorporate various machine learning algorithms to define more robust 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 [8, 9, 7] 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.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Weiner. The lorel query language for semi-structured data. In *Journal of Digital Library*, 1998.

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

[3] C. Allen. WIDL: Application Integration with XML. *World Wide Web Journal, 2(4)*, 1997.

[4] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. *Proc. ICDE'98*, Feb., 1998.

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

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

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

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

[9] 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.

[10] P. Buneman, S. Davidson, and G. H. D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM SIGMOD Conference*, 1996.

[11] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. *http://www.w3c.org/TR/1998/NOTE-xml-ql-19980819*, 1998.

[12] R. Doorenbos, O. Etsioni, and D. S. Weld. A scalable comparison-shopping agent for the world-wide-web. In *Proceedings of the First Int. Conference on Autonomous Agents*, 1997.

[13] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the world wide web. *Proceedings of Autonomous Agents*, pages 39–48, 1997.

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

[15] J. Gruser, L. Raschid, M. Vidal, and L. Bright. A Wrapper Generation Toolkit to Specify and Construct Wrappers for Web Accessible Data Sources. *ftp://ftp.umiacs.umd.edu/pub/louiqa/BAA9709/PUB98/1CoopIS98.ps*, 1998.

[16] 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.

[17] 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.

[18] G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold. Jedi: Exchanging and synthesizing information from the web. *Coopis*, 1998.

[19] T. Kistlera and H. Marais. WebL: a Programming Language for the Web. *http://www.research.digital.com/SRC/WebL/index.html*, 1998.

[20] 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.

[21] D. Konopnicki and O. Shemueli. W3qs: A query system for the world wide web. In *Proceedings of the Very Large Databases Conference*, 1995.

[22] 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.

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

[24] 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.

[25] 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.

[26] A. O. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

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

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

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

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

[31] WWWC. XML Pointer Language. *http://www.w3.org/TR/1998/WD-xptr-19980303*, 1998.

Appendix A

```
......
<Maximum_and_Minumum_Temperatures>
<Description>Maximum and Minimum Temperatures</Description>
   <Maximum_and_Minimum_Temperatures_Child>
      <Maximum_Temperature>
         <Description>MaximumTemperature F(C)</Description>
         <Value>82.0(27.8)</Value>
       </Maximum_Temperature>

      <Minimum_Temperature>
         <Description>MinimumTemperature F(C)</Description>
         <Value>62.1(16.7)</Value>
      </Minimum_Temperature>

      <TD>
         <Description></Description>
         <Value>In the 6 hours preceding Oct 29, 1998 - 06:53 PM EST / 1998.10.29 23:53 UTC</Value>
      </TD>
   </Maximum_and_Minumum_Temperatures_Child>
   <Maximum_and_Minimum_Temperatures_Child>
      <Maximum_Temperature>
         <Description>MaximumTemperature F(C)</Description>
         <Value>80.1(26.7)</Value>
       </Maximum_Temperature>

      <Minimum_Temperature>
         <Description>MinimumTemperature F(C)</Description>
         <Value>45.0(7.2)</Value>
      </Minimum_Temperature>

      <TD>
         <Description></Description>
         <Value>In the 24 hours preceding Oct 28, 1998 - 11:53 PM EST / 1998.10.28 0453 UTC</Value>
      </TD>
   </Maximum_and_Minumum_Temperatures_Child>
</Maximum_and_Minumum_Temperatures>
......
```

Figure 14: A fragment of the XML document for the NWS Savannah weather report page