IWrap: Instant Web Wrapper Generator

Aykut Firat, Denis Peleshchuk, Prakash Rao

Working Paper CISL# 2000-10
June 2000

Composite Information Systems Laboratory (CISL)
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02142

IWrap: Instant Web Wrapper Generator

Aykut Firat[1], Denis Peleshchuk[2], Prakash Rao[2]

[1]MIT Sloan School of Management
Cambridge, MA 02139
aykut@mit.edu

[2]Computer Science Department
Harvard University
Cambridge, MA 02139
dpeleshc@fas.harvard.edu, prakash.rao@tfn.com

Abstract

In this paper, we describe an automatic Web wrapper generator that creates specification files, which contain the schema information and extraction rules for a class of Web pages. These specification files can then used by a wrapper engine (e.g. MIT COIN Grenouille) to extract information from the semi-structured Web sites. We create specification files through a WYSIWYG GUI with minimal user interaction. Two different algorithms are developed that map the user input to extraction rules in terms of Regular Expressions. We also present example cases used to test the effectiveness of our two approaches.

Introduction

World Wide Web is becoming the standard environment for information publishing as the number of data sources on the Web is growing at a rapid rate. These semi-structured data sources provide an opportunity to exploit extensive valuable data in various domains. For example free data sources such as Edgar, Quicken, Zacks and Hoovers provide comprehensive financial information about US companies. These sources, however, are often displayed in HTML, which is primarily designed for human interaction through a browser. Because HTML mixes the content and the presentation format, the task of identifying and retrieving the content becomes a non-trivial task for a computer program. Wrapper engines are thus developed by various researchers that apply different algorithms to extract the requested content from these data sources.

Wrapper engines usually take as input a specification file that consists of declarative extraction rules – usually in terms of Regular Expressions- and schema information. For people, who are expert in Regular Expressions these files can be easily created probably in less than fifteen minutes in most cases. For people with little Regular Expressions expertise, it could take hours depending on the complexity of the page. For other people who do not have any Regular Expressions knowledge, the task is not at all practical and they have to rely on others. In this paper we describe IWrap, an automatic wrapper generator that generates specification files based on high-level user input, which makes spec-file generation a trivial task. The specification files produced by IWrap are directly used with the MIT COIN Group's Grenouille (Bressan 98) wrapper engine without any modification. Specification files for other engines using Regular Expressions can also be generated with little mo dification in our source code.

In the next section, we first give background information about the Grenouille wrapper engine and spec-file creation. Then we provide an operational scenario describing how IWrap GUI works. After mentioning related work, we explain the two algorithms we have used to map user inputs into extraction rules in terms of Regular Expressions. Next we provide test results based on a collection of Web pages. Finally we talk about the current limitations and future plans.

Background

In this study, we are only interested in extraction engines that mainly use Regular Expressions in their execution plan. In particular we generate spec files for the Grenouille wrapper engine. The basic architecture of the Grenouille is shown in figure 1. Grenouille, aiming to treat Web as a giant database, provides a Structured Query Language (SQL) interface to users. Each specification file is in fact a combination of the schema information and the extraction rule for a class of Web pages. An example spec file is shown in figure 2. In this spec file, the export and the relation tags provide the schema information. The second HREF tag is the combination of the address and the method used in the http connection. It is used to locate the Web page and usually requires a user input like a ticker symbol. The extraction rule, which is a pattern constructed using Regular Expressions, is expressed in the content tag. The wrapper engine uses the spec-files grouped under a database and registered with the system to extract the requested information during run time. It can be tested in

http://context.mit.edu/demos/wrapper/interface.sHTML.

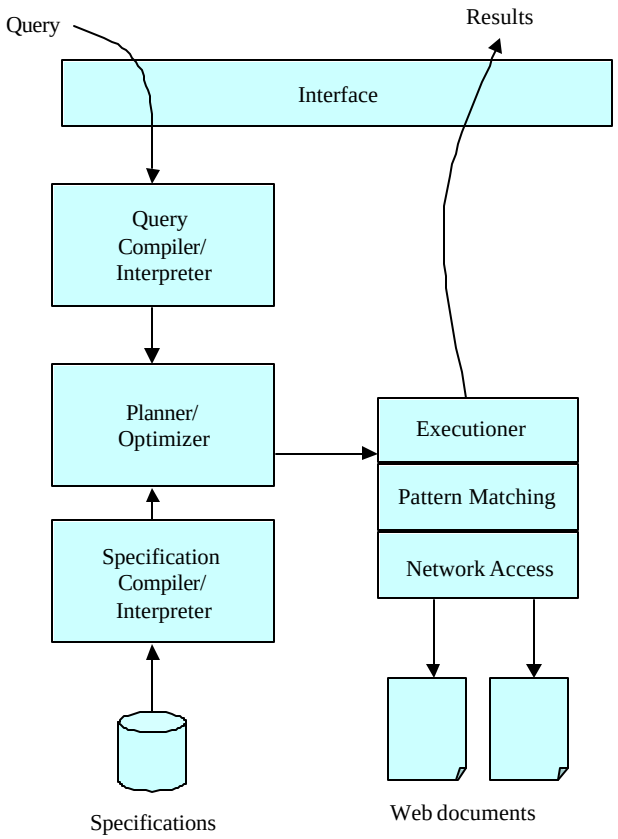

Figure 1. Grenouille Web Wrapper Architecture (Adopted from Shah 98)

```
#HEADER
#RELATION=profile
#HREF=GET http://quicken.elogic.com
#EXPORT=profile.Ticker profile.Name profile.Description
profile.Address profile.Phone #ENDHEADER
#BODY
   #PAGE
      #HREF=GET
http://www.quicken.com/investments/snapshot/?symbol=
##profile.Ticker##

   #CONTENT=Company\s*Name:[^"]*"Arial"><B>##pro
file.Name##</B></FONT></TD>.*Address:[^"]*"Arial"><
B>##profile.Address##</B></FONT></TD>.*Telephone:[
^"]*"Arial"><B>##profile.Phone##</B>.*</TD>

   #ENDPAGE
#ENDBODY
```

Figure 2. Example Spec File for Company Profiles

Operational Description

Grenouille is a quite flexible system and people are usually impressed with its abilities. We have used it successfully in Personal Investor Wizard, a Java application that integrates financial information from various Web sources. Grenouille can also be executed from Microsoft Excel directly and displays the query results in cells. In our past demonstrations, the invariant question has been: We want to use it but what do we need to know? The troublesome answer included the fact that they had to either know Regular Expressions or rely on people who know them.

With IWrap users no longer need to know Regular Expressions. In addition IWrap will save considerable amount of time in creating spec files even for the experts. We admit that there are situations, which IWrap cannot handle currently but with some more additional work they will be eliminated.

Although IWrap automatically creates spec files, the users still need to provide some minimal input. We next describe how the user interacts with the system. Please refer to the snaphots of IWrap shown in figure 3 for the following operational scenario:

After starting IWrap users enter the URL of the page they want to wrap. The URL in most cases will include input parameters such as ticker symbols, search texts, etc. Page name and the input attributes are automatically inserted into the input table. Input table also contains the relation attribute whose value is expected from the user. Another attribute in the input table is method, which determines the method used in connecting to the page. The default value is GET, but the user may choose to change it to POST.

In the output table the user first provides a header value, which must be unique in the page. In future versions we will not require this uniqueness property, but currently it simplifies some of the technical difficulties caused by the ASCII file location-mapping problem from the HTML renderer that uses gap content text style. Next the user highlights the text he/she wants to extract and also provides a name for it. The name can be any word and does not have to exist in the page. The name will later be used by the user in issuing an SQL query. We also provide a source code pane that can be used when the information to be extracted is hidden in an HTML tag.

IWrap lets users edit the table cells manually, or supply a highlighted information to a selected cell in addition to adding entire rows. The user can also remove the unwanted rows and add new ones very easily.

When the user supplies all the information and clicks on the auto wrap button, the spec file creation will begin and system messages will be shown in the messages window. If the creation is successful the user can immediately start issuing SQL queries to the wrapper engine.

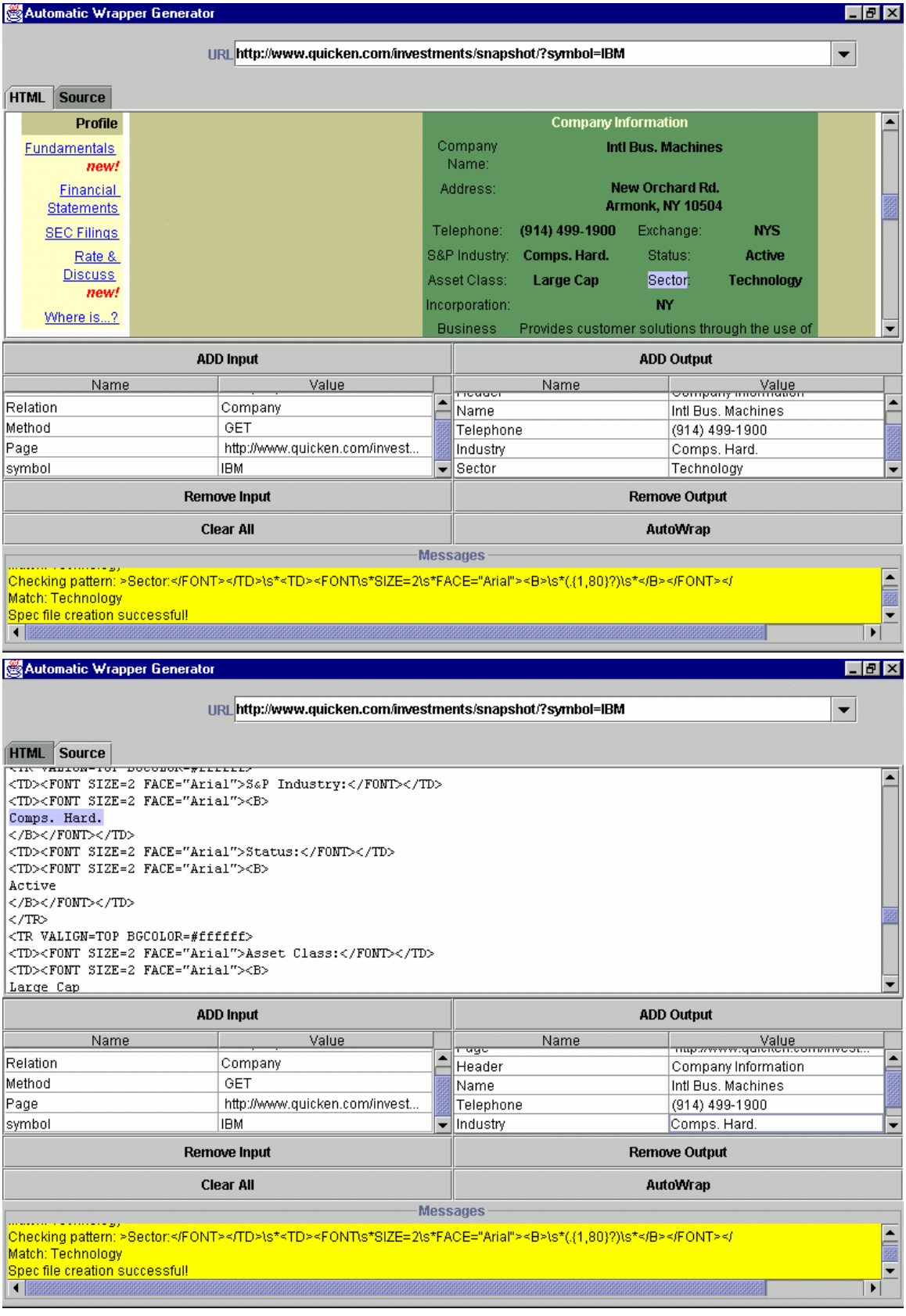

Figure 3. IWrap Snapshots

Related Work

Although there is substantial work done in the Web wrapping area, there is little done for automatic spec file generation. For instance (Azavant and Sahuguet 98) discuss different approaches to Web wrapping by classifying them into two groups: one who view a Web document only as a flow of tokens and completely ignore the tag-based hierarchy (Grenouille is classified into this group in their paper) and one who take advantage of the hierarchical structure implied by the HTML tags (their W4F engine is classified into this category.) They also describe a declarative specification language, but there is not any reported work about whether spec files can be created automatically with no knowledge of their language.

(Ahish and Knoblock 97) present their work on semi-automatic wrapper generation, but they concentrate only on several sources with very fixed structure. Also, their major goal is not the actual wrapper generation, but finding structure and possible attributes in a document. In our case searching for possible attributes, although a challenging problem, does not add much value because the user is willing to select the desired attributes on a page.

Spec File Generation

We used two different approaches in generating spec files, which are explained below.

Gradual Expansion and Database Lookup Approach

This approach uses two different algorithms to come up with Regular Expressions that can be used to automatically extract values. Because they are relatively independent, they are applied in turn until a solution is found. These two methods are described below:

Database Lookup

Many Web data sources use common patterns to represent data. An example of such a pattern is HTML table when an attribute's value is in the next column after a header. For example:

```
<table>
…
<td>Sales</td><td>1000</td>
…
</table>
```

In this case, "Sales" is the header and "1000" is its value that we want to extract. Typically, this information is generated in the same way for different companies that have different values for Sales.

To implement this approach, we collect common patterns and wrap them into Regular Expressions. We store two sets of expressions, one for attributes with single value and one for attributes with multiple values. In both cases, we try to apply stored Regular Expressions in turn until the right match is found. OroMatcher, a Perl5 Regular Expression compatible package for Java, is used for Regular Expression matching.

Some sample patterns in our database are shown below:

```
>##Header##(?:<[^>]*>)+([^<]*)<
```

This pattern will match cases when a value is the first non-tag text after a header. Since HTML tags are not displayed, this means that the value follows the header when HTML is rendered.

```
<li>\s*<a href="[^"]*">(.*?)</a>
```

This pattern matches multiple values that are represented by the HTML list.

Advantages of this approach can be listed as:

- Produced Regular Expressions can be clearly explained and thus they are relatively understandable by humans.
- They are robust, because their reliance on small HTML details is minimal. For example, in a pattern that relies on HTML tables, all tags that are inside <td> and </td> are ignored. Format tags such as font can be changed, but the pattern will still work.
- Easily extensible by adding new patterns to the database.

One disadvantage of this method is that it requires the participation of a human expert in the initial population of the database. Also if a match cannot be found in the database, this method will not try anything else and fail.

The flowchart diagram shown in figure 4 demonstrates the simplified version of the database lookup algorithm.

Sample pages that are processed successfully with this method are:

- Yahoo company profile page:

http://biz.yahoo.com/p/i/ibm.HTML.

Company address, phone, number of employees, etc. are extracted for any given company.

- Quicken stock quote page:

http://www.quicken.com/investments/quotes/?symbol=ibm

Last Trade, Change, Day's Range, etc. are extracted for any given ticker symbol.

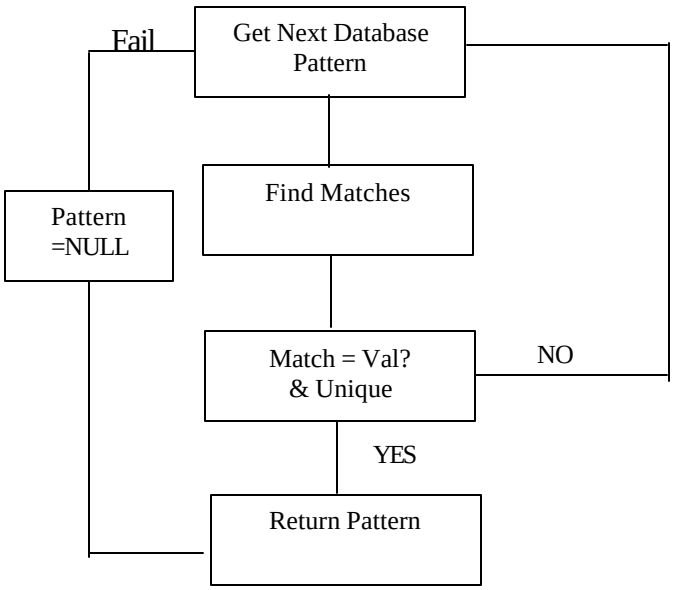

Figure 4. Database Lookup Flowchart

Gradual Expansion Algorithm

This method can be used when data in an HTML page is not represented by any common pattern, but can still be uniquely identified by the surrounding data items. The algorithm is slightly different if the attribute has single or multiple values. [For example: Company name attribute is single valued, but company news can be multi-valued]

In single value cases, we start with the text immediately to the left and immediately to the right of the attribute value and expand it one word (actually we expand until the next delimiter) at a time until we find a pattern that is unique in the document. Expansion to the left is done twice as fast as to the right because unique patterns are more frequently found to the left according to our experience.

The advantage of this method is that it can find a solution in practically all cases. There are, however, several important weaknesses:

- Inflexibility: Excessive reliance on underlying HTML leads to inflexible patterns that can stop working even after small changes in HTML.
- Overfitting: Only one sample page in used, and thus a pattern found can be acceptable only for this specific page, but not for other similar pages. For example, it can work for extracting stock quote for IBM, but not for Microsoft from the same data source.

There is not much that can be done about the first problem. A possible improvement would be handling at least most frequently changed items in HTML. For example, instead of matching a font tag with specific font face and size, we could match a font tag with any font face or size. This makes discovered pattern more robust, but on the other hand it makes it more difficult to find a pattern that uniquely identifies selected value.

The second problem can be addressed by using more than one page as input when finding a pattern. For example, we can use stock quote pages for both IBM and Microsoft when we want to extract stock quotes. We are in the process of implementing this feature which will request test inputs from the user.

Same problems exist for multiple value cases, but the last problem is not as serious because there are multiple test cases on the same page.

The algorithm in this case starts by expanding the text to the immediate left of all selected attribute values. If the text is different, it is replaced by a Regular Expression like ".{0, n}"- which means 0 to n occurrences of any symbol- and expansion continues. If the text is the same for all selected values, we check whether that pattern can uniquely identify selected values in the document.

It is clear that this approach will work only when similar HTML tags precede selected attribute values. Another restriction is that they have to be followed by the same HTML tag. These restrictions generally should not cause problems because multiple values for the same attribute typically have common surrounding data items.

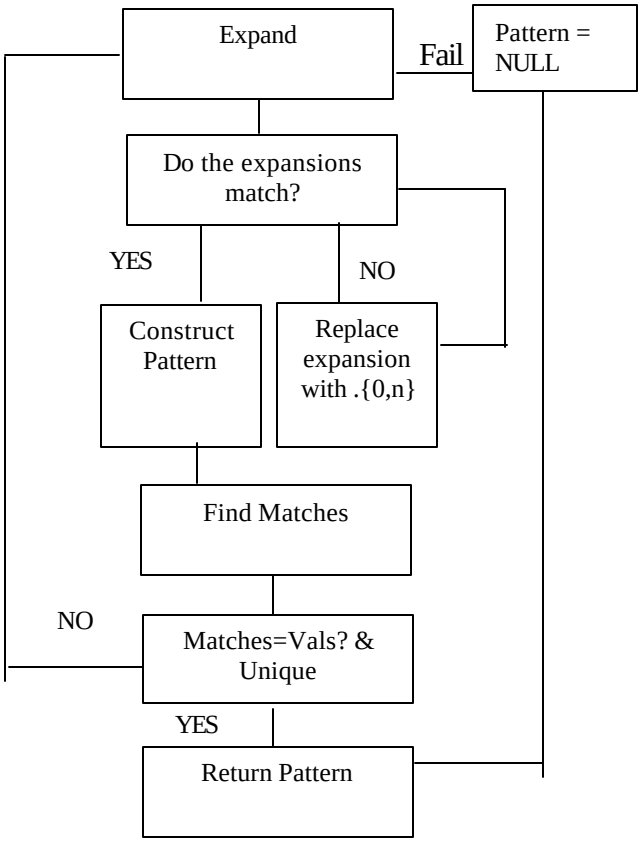

Figure 5. Gradual Expansion Algorithm

Sample pages for which this approach produced good results are:

- Quicken company comparison: http://www.quicken.com/investments/comparison/?symbol=IBM&origin=quotes.
Competitors, company names and market capitalization values are extracted for any given stock ticker.
- Hoovers company information page: http://www.hoovers.com/capsules/10796.HTML?ticker.
Top competitors are extracted for any given ticker.

The second page is a good example of how this method works. Html for top competitors looks like this:

<A HREF="/capsules/10381.HTML">Compaq</A>

It is similar for all of them, but file names (10381.HTML in this case) are different for different companies. The algorithm finds the following solution in this case:

"/capsules.{0,18}?>##test.Val:([^<>]?)##</A>*

The current algorithm is somewhat simplistic because when it finds a difference in expanded text between selected values, it simply replaces the text that has been extended up to that time with ".{0,n}". An improvement could be made to replace only the part that is different with a Regular Expression. In many cases, a unique pattern could be found faster if this technique is used. In the example above, a better pattern would be:

"/capsules/.{0,10}?\.HTML">

The multiple value case suffers from the same problem: excessive reliance on HTML. Similar solutions as described above to make patterns more general for the single value case, also apply to the multiple value case.

As mentioned earlier, another limitation is that the surrounding data items have to have exactly the same structure for all values that need to be extracted. For example, the following HTML from www.mit.edu "Of Interest" section cannot be properly processed by this approach (it should be noted, however, that it can be handled by the database method):

* IEEE publications*
* MIT Party/Events Listing*
* Help Pages*

There are three different values here: "IEEE publications", "MIT Party/Events Listing", and "Help Page". The structure of the surrounding HTML is similar for all three of them, but unlike the case with top competitors above, paths inside href tags have different directory structure. By simply expanding to the next delimiter, we fail to find a common pattern. This could be handled by further abstractions, i.e. considering anything inside double quotes in href as one token and we are in fact in the process of implementing this idea.

Token Approach

Solving the problem of automatic wrapper generation for semi-structured Web pages is a fairly complex and difficult task. Hence, it warrants an investigation of a number of possible approaches to finding a good solution. For an earlier project (Personal Investor Wizard), we had written approximately 20-30 odd spec files manually, and were very successful in extracting values for attributes from those Web pages. A possible question that can be raised is: Are there any common properties in all these spec files, that we might somehow be able to incorporate for automatic wrapper generation?

To answer this question, we conducted some analysis (or perhaps we should call it manual data mining) on our previous body of work on spec files. We came up with the following interesting observations:

1. No pattern in the #CONTENT section of a spec file ever started with a close-tag, i.e. a tag of the form </…>
2. If a pattern had some special kinds of tags, like a <A HREF=…> type tag for example, then some processing needed to happen on the contents of this tag itself.
3. Most patterns (not all though) seem to end with one or more from a list of popular close-tags, alone or in sequence. Some of the tags in this list include tags such as </A>, </TD>, </B>, </TR>, </FONT>, </STRONG>, </P>, </TITLE> etc.

We concluded that we can use these heuristics to come up with another approach to solving this problem.

Token algorithm

The algorithm we came up with has basically the following steps:

1. Parse the HTML source into a list of tokens, where a token could be a tag, non-tag text, or spaces between tags. For example, if the input HTML looked something like:

<HTML>foo<TITLE>title</TITLE><BODY>body</BODY> </HTML>

Then the output tokens would be:

- *<HTML>*
- *foo*
- *<TITLE>*
- *title*
- *</TITLE>*
- *<BODY>*
- *body*
- *</BODY>*
- *spaces*
- *</HTML>*

2. Apply the heuristics to come up with a candidate pattern.
3. Generalize the candidate pattern by turning it into a Regular Expression.
4. Generate the spec file.

HTML Parsing

As of the time this paper was being written, the token algorithm was not integrated to our GUI, but we tested it by reading all the desired input from a simple text file. Once the sample HTML source is read, parsing is done to break it up into tokens as we described. From an implementation standpoint, each token is stored as an object that has 2 fields: *a String value, and a Boolean* to indicate if it's a tag or not. Then all the token objects are stored in a vector structure. We also create a separate vector that stores a list of popular close-tags, as specified earlier.

Pattern Creation

Three of the input parameters that need to be supplied are:

- ***attributeText=*** the actual text in the HTML source that represents the particular attribute
- ***attributeName=*** the name we want to give this attribute in our wrapper spec file
- ***multiValue= (yes|no)***. To indicate whether the attribute has multiple values or not.

Given the above information, the pseudo-code is:

- Traverse the vector of token objects until you reach the attributeText value.
- *If multiValue=no, then start the pattern string with this attributeText, else move to the next token.*
- If multiValue=yes, skip space tokens, else concatenate the spaces as part of the pattern string.
- *If multiValue=yes , skip all tokens that are close-tag types (of the form </…>), else concatenate these to the pattern string.*
- *If multiValue=yes, skip some special tokens that are most likely associated with the attributeText. Some examples are
 or <UL>. If multiValue=no however, keep concatenating these to the pattern string.*
- *Concatenate all tags after the attributeText until you reach the first non-tag text token.*
- *Check if this non-tag text is something like ":" or " " or "=" etc. i.e. check from a list that would definitely NOT qualify as an attribute value. While this condition is true, concatenate these to the pattern string and keep moving to the next token. Also if the following tokens are tags again, follow the same procedure.*
- *Now you have reached a non-tag text token that we assume is most likely to be the attribute value. Replace this with the ##…## format as specified by the Grenouille wrapper engine.*
- *Then keep reading and concatenating the tokens that follow the attribute value, until you reach a token that is in the list of popular close-tags.*
- *As there may be several popular close-tags in sequence, keep reading and concatenating to the pattern string, until you reach a token that is NOT a popular close-tag.*

Generalization

Now that we have a candidate pattern string, we are ready to wrap it as a Regular Expression. Here again we process the entire candidate pattern string on a token by token basis. Tokens are processed differently, depending on whether the multiValue attribute is set to true or not. Some more complex tokens like <A HREF=…> need to be handled separately. We follow all the rules for Regular Expressions. For example, spaces need to be replaced as \s* and a literal '?' character needs to be replaced as "\?" etc. When we are done with all the tokens in the candidate pattern, we are ready to generate the wrapper specification file.

Spec File Generation

Based on some of the other input, such as the relation name, method to be used, provided in the input text file we can easily generate a specification file, following the rules for specification files, as specified by the Grenouille wrapper engine.

One of the advantages to this approach is that because everything is stored as tokens in a Java vector structure, it is possible to do some forward or backward lookups for comparison purposes, when processing some special tokens. For example, processing a <A HREF=…> type token can be done by doing a forward lookup to the next <A HREF=…> token, comparing the two, and replacing only the parts that are different with a Regular Expression. This greatly facilitates generating patterns, especially for cases where one attribute has multiple values.

One of the disadvantages of this approach is that we are relying primarily on our past body of manually generated

wrapper files, which certainly do not cover all the different kinds of HTML tags. However, the assumption here is that these Web sources are all semi-structured. Most of these semi-structured Web sources have similar presentations, i.e. rather than using a whole lot of new tags, they all seem to use the same subset of tags repeatedly. Besides extending this to include more tags is relatively easy. As we test with more cases, we simply add handling for any special tags that we have not yet covered.

We tested this approach on some different Web sources, and as expected, achieved different results depending on the structure and simplicity of the Web source in question. Some sample pages where we had success were:

- http://www.hoovers.com/cgi-bin/search.cgi?search=symbol&query=ibm.

Competitors were extracted for any ticker.

- http://biz.yahoo.com/research/indgrp/.

Different industry names were extracted for any given company.

- http://www.mit.edu.

Items of interest were extracted.

- http://www.odci.gov/cia/publications/factbook/in.HTML

Location attribute was extracted.

Some sample pages where we had problems were:

- http://www.quicken.com/investments/quotes/?symbol=ibm.

Had a problem extracting the Last Trade attribute.

- http://biz.yahoo.com/p/i/ibm.HTML.

Had a problem extracting the complete address, from the address attribute. Only part of the address was extracted.

Some of the problems are due to the fact that the assumptions we make about the format of the attribute values v/s what actually exists in some HTML sources do not match. For example, if the attribute value is split up in the following form:

<B>230<FONT SIZE=1>3/8</FONT>

instead of just being one single piece of text in between tags, then we run into problems.

Conclusions and Future Work

In this paper we have described two different approaches to automatic spec file generation through a WYSIWYG user interface. We are quite happy with the initial test results. Our results demonstrate that our approach to automatic wrapper generation can be very useful when used together with a complete Web wrapper/mediation package. It gives non-technical users easy access to Web wrapper technology.

Our methods still have some limitations, but by spending more time on testing with different Web data sources, we believe they can be tuned to handle all practical cases.

To make our approach even more user-friendly, we are planning to integrate it with a wrapper/mediation engine written in Java. Instead of using the engine provided by COIN group, we will develop a complete package that can be installed on any user's desktop easily. That package should provide all the tools necessary for easy maintenance of multiple Web data sources. Because we have the ability to automatically generate wrapper specification files, we can do away with one of the biggest maintenance problems by detecting changes in data sources and regenerating spec files. This process can be easily fully automated.

Example application areas that can benefit from this technology are:

- Shopping agents: finding best prices at different Web stores.
- Financial research applications: integrating financial information from different Web data sources.

Data extraction can either be done in real time or, in cases when performance is important, in batch processes that update conventional relational databases with information collected from the Web.

We already started working on some of these ideas and will release a new version of our system in a short time.

References

Ashish, N., Knoblock, C. 1997. Wrapper Generation for Semi-structured Internet Sources. *In International Workshop on Management of Semistructured Data, 1997.*

Azavant F, Sahuguet, A. 1998. W4F: a WysiWyg Web Wrapper Factory for Minute-Made Wrappers, Forthcoming.

Bressan, S., 1998. Grenouille Version 1.1, Forthcoming.

Shah , S. A. Z. 1998. Design and Architecture of the Context Interchange System. Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT.