Extracting Semistructured Information from the Web

J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo

Department of Computer Science
Stanford University
Stanford, CA 94305-9040

{hector,joachim,cho,aranha,crespo}@cs.stanford.edu
http://www-db.stanford.edu/

Abstract

We describe a configurable tool for extracting semistructured data from a set of HTML pages and for converting the extracted information into database objects. The input to the extractor is a declarative specification that states where the data of interest is located on the HTML pages, and how the data should be "packaged" into objects. We have implemented the Web extractor using the Python programming language stressing efficiency and ease-of-use. We also describe various ways of improving the functionality of our current prototype. The prototype is installed and running in the TSIMMIS testbed as part of a DARPA I^3 (Intelligent Integration of Information) technology demonstration where it is used for extracting weather data form various WWW sites.

1. Introduction

The World Wide Web (WWW) has become a vast information store that is growing at a rapid rate, both in number of sites and in volume of useful information. However, the contents of the WWW cannot be queried and manipulated in a general way. In particular, a large percentage of the information is stored as static HTML pages that can only be viewed through a browser. Some sites do provide search engines, but their query facilities are often limited, and the results again come as HTML pages.

In this paper, we describe a configurable extraction program for converting a set of hyperlinked HTML pages (either static or the results of queries) into database objects. The program takes as input a specification that declaratively states where the data of interest is located on the HTML pages, and how the data should be "packaged" into objects. The descriptor is based on text patterns that identify the beginning and end of relevant data; it does not use "artificial intelligence" to understand the contents. This means that our extractor is efficient and can be used to analyze large volumes of information. However, it also means that if a source changes the format of its exported HTML pages, the specification for the site must be updated. Since the specification is a simple text file, it can be modified directly using any editor. However, in the future we plan to develop a GUI tool that generates the specification based on high-level user input.

The vast majority of information found on the WWW is *semistructured* in nature (e.g., TSIMMIS [1], LORE [2], Garlic [3], Information Manifold [4], RUFUS [5]). This means that WWW data does not have a regular and static structure like data found in a relational database. For example, if we look at classified advertisements on the Web, the "fields" and their nesting may differ across sites. Even at a single site, some advertisements may be missing information, or may have extra information. Because of the semistructured nature of WWW data, we have implemented our extractor facility so that it outputs data in OEM (Object Exchange Model) [1] which is particularly well suited for representing semistructured data. OEM is the model used by our TSIMMIS (The Stanford IBM Manager of Multiple Information Sources) project. Thus, one of our TSIMMIS wrappers [6] can receive a query targeted to a set of HTML pages. The wrapper uses the extractor to retrieve the relevant data in OEM format, and then executes the query (or

whatever query conditions have not been applied) at the wrapper. The client receives an OEM answer object, unaware that the data was not stored in a database system.

In this paper, we describe our approach to extracting semistructured data from the Web using several examples. Specifically, we illustrate in detail how the extractor can be configured and how a TSIMMIS wrapper is used to support queries against the extracted information.

2. A Detailed Example

For our running example, let us assume that we have an application that needs to process weather data, such as temperature and forecast, for a given city. As one of its information sources, we want to use a Web site called IntelliCast [7] which reports daily weather data for most major European cities (see Figure 1).

		Tue, Jan 28, 1997		*Wed, Jan 29, 1997*	
country	***city***	***forecast***	***hi/lo***	***forecast***	***hi/lo***
Austria	Vienna	snow	-2/-7	snow	-2/-7
Belgium	Brussels	ptcldy	3/-4	ptcldy	3/-4
Czech Republic	Prague	snow	-1/-7	snow	-1/-7
Denmark	Copenhagen	fog	3/-1	fog	3/-1
England	Birmingham	ptcldy	9/-3	ptcldy	7/3
England	Liverpool	ptcldy	8/2	ptcldy	6/2
England	London	ptcldy	9/0	ptcldy	8/4
England	Manchester	ptcldy	8/-1	ptcldy	6/3
England	Plymouth	ptcldy	9/3	ptcldy	8/5

Figure 1: A snapshot of a section of the IntelliCast weather source.

Since this site cannot be queried directly from within another application (e.g., "What is the forecast for Vienna for Jan. 28, 1997?") we first have to extract the contents of the weather table from the underlying HTML page[1] which is displayed in Figure 2.

2.1 The Extraction Process

Our *configurable extraction program* parses this HTML page based on the specification file shown in Figure 3. The specification file consists of a sequence of *commands*, each defining one extraction step. Each command is of the form

[variables, source, pattern]

where *source* specifies the input text to be considered, *pattern* tells us how to find the text of interest within the source, and *variables* are one or more extractor variables that will hold the extracted results. The text in variables can be used as input for subsequent commands. (If a variable contains an extracted URL, we can also specify that the URL be followed, and that the linked page be used as further input.) After the last command is executed, some subset of the variables will hold the data of interest. Later we describe how the contents of these variables are packaged into an OEM object.

[1] The line numbers shown on the left-hand side of this and the next figures are not part of the content but have been added to simplify the following discussions.

```
 1 <HTML>
 2 <HEAD>
 3 <TITLE>INTELLICAST: europe weather</TITLE>
 4 <A NAME="europe"></A>
 5 <TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=509>
 6 <TR>
 7 <TD colspan=11><I>Click on a city for local forecasts</I><BR></TD>
 8 </TR>
 9 <TR>
10 <TD colspan=11><I> temperatures listed in degrees celsius </I><BR></TD>
11 </TR>
12 <TR>
13 <TD colspan=11><HR NOSHADE SIZE=6 WIDTH=509></TD>
14 </TR>
15 </TABLE>
16 <TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=514>
17   <TR ALIGN=left>
18     <TH COLSPAN=2><BR></TH>
19     <TH COLSPAN=2><I>Tue, Jan 28, 1997</I></TH>
20     <TH COLSPAN=2><I>Wed, Jan 29, 1997</I></TH>
21   </TR>
22   <TR ALIGN=left>
23     <TH><I>country</I></TH>
24     <TH><I>city</I></TH>
25     <TH><I>forecast</I></TH>
26     <TH><I>hi/lo</I></TH>
27     <TH><I>forecast</I></TH>
28     <TH><I>hi/lo</I></TH>
29   </TR>
30   <TR ALIGN=left>
31     <TD>Austria</TD>
32     <TD><A HREF=http://www.intellicast.com/weather/vie/>Vienna</A></TD>
33     <TD>snow</TD>
34     <TD>-2/-7</TD>
35     <TD>snow</TD>
36     <TD>-2/-7</TD>
37   </TR>
38   <TR ALIGN=left>
39     <TD>Belgium</TD>
40     <TD><A HREF=http://www.intellicast.com/weather/bru/>Brussels</A></TD>
41     <TD>fog</TD>
42     <TD>2/-2</TD>
43     <TD>sleet</TD>
44     <TD>3/-1</TD>
45   </TR>
       .
       .
    </TABLE>
       .
  </HTML>
```

Figure 2: A section of the HTML source file.

Looking at Figure 3, we see that the list of commands is placed within the outermost brackets '[' and ']', and each command is also delimited by brackets. The extraction process in this example is performed by five commands. The initial command (lines 1-4) fetches the contents of the source file whose URL is given in line 2 into the variable called `root`. The '#' character in line 3 means that everything (in this case the contents of the entire file) is to be extracted. After the file has been fetched and its contents are read into `root`, the extractor will filter out unwanted data such as the HTML markup commands and extra text with the remaining four commands.

The second command (lines 5-8) specifies that the result of applying the pattern in line 7 to the source variable `root` is to be stored in a new variable called `temperature`. The pattern can be interpreted as follows: "discard everything until the first occurrence of the token `</TR>` ('*' means discard) in the second table definition and save the data that is stored between `</TR>` and `</TABLE>` ('#' means save)." The two `<TABLE` tokens between the '*' are used as navigational help to identify the correct </TR> token since there is no way of specifying a numbered occurrence of a token (i.e., "discard everything until the *third* occurrence of `</TR>`"). After this step, the variable `temperature` contains the information that is stored in lines 22 and higher in the source file in Figure 2 (up to but not including the subsequent `</TABLE>` token which indicates the end of the temperature table).

```
 1 [["root",
 2   "get('http://www.intellicast.com/weather/europe/')",
 3   "#"
 4  ],
 5  ["temperatures",
 6   "root",
 7   "*<TABLE*<TABLE*</TR>#</TABLE>*"
 8  ],
 9  ["_citytemp",
10   "split(temperatures,'<TR ALIGN=left>')",
11   "#"
12  ],
13  ["city_temp",
14   "_citytemp[1:0]",
15   "#"
16  ],
17  ["country,c_url,city,weath_tody,hgh_tody,low_today,weath_tomorrow,hgh_tomorrow,low_tomorrow",
18   "city_temp",
19   "*<TD>#</TD>*HREF=#>#</A>*<TD>#</TD>*<TD>#/#</TD>*<TD>#</TD>*<TD>#/#*"
20  ]]
```

Figure 3: A sample extractor specification file.

The third command (lines 9-12) instructs the extractor to split the contents of the `temperature` variable into "chunks" of text, using the string `<TR ALIGN=left>` (lines 22, 30, 38, etc. in Figure 2) as the "chunk" delimiter. Note, each "chunk" represents one row in the temperature table. The result of each split is stored in a temporary variable called `_citytemp`. The underscore at the beginning of the name `_citytemp` indicates that this is a temporary variable; its contents will not be included in the resulting OEM object. The split operator can only be applied if the input is made up of equally structured pieces with a clearly defined delimiter separating the individual pieces. If one thinks of extractor variables as lists (up until now each list had only one member) then the result of the split operator can be viewed as a new list with as many members as there are rows in the temperature table. Thus from now on, when we apply a pattern to a variable, we really mean applying the pattern to *each* member of the variable, much like the `apply` operator in Lisp.

In command 4 (lines 13-16), the extractor copies the contents of each cell of the temporary array into the array `city_temp` starting with the second cell from the beginning. The first integer in the instruction `_citytemp[1:0]` indicates the beginning of the copying (since the array index starts at 0, 1 refers to the second cell), the second integer indicates the last cell to be included (counting from the end of the array). As a result, we have excluded the first row of the table which contains the individual column headings. Note, that we could have also filtered out the unwanted row in the second command by specifying an additional `*</TR>` condition before the '#' in line 7 of Figure 3. The final command (lines 17-20) extracts the individual values from each cell in the `city_temp` array and assigns them into the variables listed in line 17 (`country`, `c_url`, `city`, etc.).

After the five commands have been executed, the variables hold the data of interest. This data is packaged into an OEM object, shown in Figure 4, with a structure that follows the extraction process. OEM is a schema-less model that is particularly well-suited for accommodating the semistructured data commonly found on the Web. Data represented in OEM constitutes a graph, with a unique root object at the top and zero or more nested subobjects. Each OEM object (shown as a separate line in Figure 4) contains a label, a type, and a value. The label describes the meaning of the value that is stored in this component. The value stored in an OEM object can be atomic (e.g., type *string*, *url*), or can be a set of OEM subobjects. For additional information on OEM, please refer to [8].

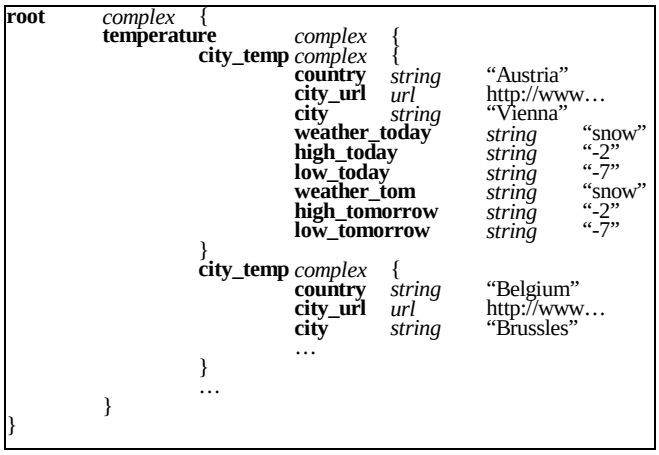

Figure 4: The extracted information in OEM format.

Notice that the sample object in Figure 4 reflects the structure of our extractor specification file. That is, the root object of the OEM answer will have a label `root` because this was the first extracted variable. This object will have a child object with label `temperature` because this was the second variable extracted. In turn, the children are the `city_temp` objects extracted next, and so on. Notice that variable `_citytemp` does not appear in the final result because it is a temporary variable.

2.2 Customizing the Extraction Results

As discussed in the previous section, the outcome of the extraction process is an OEM object that contains the desired data together with information about the structure and contents of the result. The contents and structure of the resulting OEM object are defined in a flexible way by the specification file. For instance, we could have chosen to extract additional data, and to create an OEM result that has a different structure than the one shown in Figure 4. For example, we can also extract the date values in lines 19 and 20 of Figure 2. Then, we can group together the temperature and weather data that is associated with each date, creating an OEM object such as the one depicted in Figure 5. Although not shown in our example, we could have also specified that different label names be used in the OEM object than those that are used for the extraction variables.

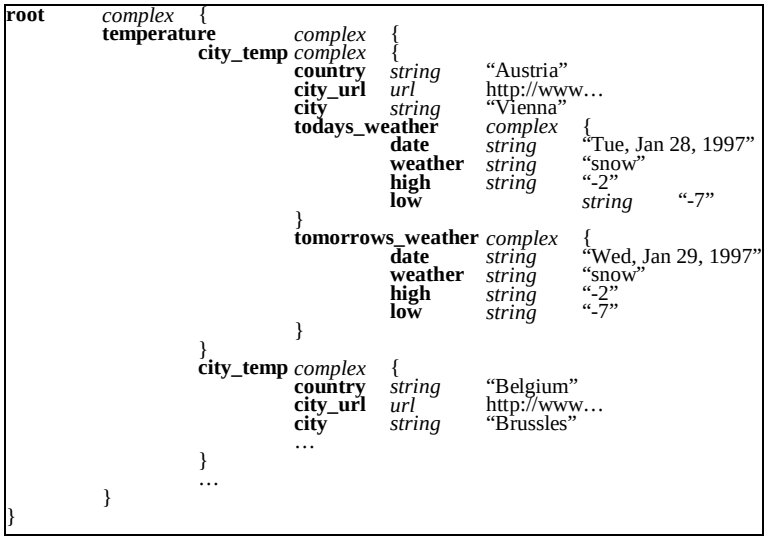

Figure 5: A different OEM result object.

It is important to note that there may be several different ways of defining the individual extraction steps that ultimately result in the same OEM answer object. Thus when defining the specification file one can proceed in a way that is most intuitive rather than worrying about finding the only "correct" set of steps. For instance, in our example, we could have avoided the usage of the temporary array `_citytemp` by filtering out the unwanted header information in the previous step. However, both approaches ultimately lead to the same result[2] (with slight differences in performance).

2.3 Additional Capabilities

In addition to the basic capabilities described in the previous section, our extractor provides several other features and constructs that simplify the extraction steps and at the same time enhance the power of the extractor. For example, the `extract_table` construct allows the automatic extraction of the contents of an HTML table (i.e., the data that is stored in each of its rows) as long as the table can be uniquely identified through some patterns in the text (this would allow us to collapse steps 2 and 3 in our example). An other useful operation is the `case` operator that allows the user to specify one or more possible patterns that are expected to appear in the input. This is especially useful for sources where the structure of the file is dynamic (in addition to the actual data). If the first pattern does not match, the parser will try to match each of the alternate patterns until a match has been found. If none of the patterns match, the parser will ignore the rest of the current input and continue parsing the data from the next input variable (if there is one).

As a last example of the extraction capabilities of our parser, consider the frequent scenario where information is stored across several linked HTML pages. For example, one can imagine that the weather data for each city is stored on its own separate Web page connected via a hyperlink. In this case, one can simply extract the URL for each city and then obtain the contents of the linked page by using the `get` operator as shown in the second line of Figure 3.

2.4 Querying the Extracted Result

In order to allow applications to query the extracted results, we need to provide a queryable interface that can process queries such as "What is the high and low temperature for Jan. 29 for Vienna, Austria?" and return the desired result (e.g., "high: -2, low -7"). Rather than developing a new query processor from scratch, we decided for now to reuse the wrapper generation tools that we have developed in the TSIMMIS project. With this toolkit, we can generate wrappers that have the ability to support a wide variety of queries that are not natively supported by the source, in our case the extracted output (for more details on wrappers see [6, 9]). With this approach, we only need to provide a simple interface that accepts a request for the entire extracted result (this is equivalent to supporting a query such as "`SELECT * FROM …`"). Making use of the wrapper's internal query engine, we can indirectly support most of the commonly asked queries (i.e., selections and projections on the extracted output). Currently, applications can interact with TSIMMIS wrappers using one of two query languages (LOREL[3] [2] and MSL[4] [10]). Wrappers return results in OEM format.

In the future, we plan to store extracted OEM results in LORE (Lightweight Object Repository) to make use of its more sophisticated query processor that has been optimized for answering (LOREL) queries on semistructured data. In addition, we will get the ability to cache extracted results which will allow us to reuse already extracted information and provides independence from unreliable sources.

[2] We chose the approach described here since it demonstrates the additional capabilities of the extractor but the solution without the temporary variable is more efficient.

[3] LOREL (LORE Language)is a query language that was developed at Stanford as part of the LORE (Lightweight Object Repository) project for expressing queries against semistructured data represented in OEM.

[4] MSL (Mediator Specification Language) is a rule-based language, which was developed as part of the TSIMMIS project for querying OEM objects.

3. Evaluation

An important design goal when developing our extractor was to find the right balance between its inherent capabilities on one hand and ease of use on the other. We think we have achieved both, which becomes apparent when one compares our approach to other existing tools such as YACC [11] or PERL [13] regular expressions, for example. Although YACC is a more powerful and more generic parser, a YACC grammar for extracting Web data from the HTML source in Figure 2 would be much more complex and difficult to generate (after all, writing a YACC specification is nothing else than writing a program using a much more complex language). For example, YACC does not support hierarchical splitting, an important feature of our extractor that demonstrates its close relationship to the hierarchical organization of Web data and simplifies the specification that a user has to write.

We have also considered using an existing HTML parser which is natively available in Python[5]. This HTML parser "understands" SGML syntax and can automatically identify HTML tags in the input stream. Upon encountering an HTML tag, the parser executes a user-defined function using the tag attributes as function input. Thus, it is very easy to extract text and HTML tags from an input file. However, the native parser does not understand the semantic connections that exist between some of the tags (i.e., begin and end tags, list members, etc.) and cannot easily be used for building an OEM object that preserves the hierarchical structure of the data. All of the processing and construction has to be handled by externally invoked user-defined functions. In addition, although this native parser is extremely flexible in terms of input processing capabilities, it is not as efficient in terms of raw processing speed as our own parser which is implemented on top of the Python `find` command[6].

A drawback of our approach is that the extraction mechanism depends on outside (human) input for describing the structure of HTML pages. This becomes an issue when the structure of source files changes rapidly requiring frequent updates to the specification file. Using a different approach, Ashish et al. [12], attempt to insert machine learning techniques into their extraction program for automatically making intelligent guesses about the underlying HTML structure of a Web site. Their approach is aimed at eliminating most of the user intervention from the extraction process. By contrast, the approach that we are pursuing here is two-pronged and relies on human intelligence supported by a flexible extractor program: (1) we are enabling our extractor to exit gracefully from a variety of cases in which the underlying structure does not match the specification, and (2), we are making the process of writing the extraction specification itself as easy and efficient as possible. In the future, we intend to develop a GUI for helping users generate and maintain correct specification files. Our planned interface will resemble a Web browser in that it can render the marked-up HTML source. More importantly however, it will enable the user to simply "highlight" the information that is to be extracted directly on the screen without having to write a single line of specification (which will be generated automatically by the GUI).

4. Conclusion

There has been much interest recently in moving data from the WWW into databases, of one type or another. This way, data that is embedded in HTML documents can be searched more effectively, and can be better managed. Our extractor is a flexible and efficient tool that provides a currently missing link between a lot of interesting data (which resides on the Web) and the applications (which have no direct access to the Web data).

We are currently using the extractor in our TSIMMIS testbed for accessing weather and intelligence data from several Web sites. As a result of our initial tests, we implemented the `case` operator as described in Sec. 2.3. The need for such an operator arose since we frequently encounter minor irregularities in the structure of the underlying HTML pages (e.g., weather data that is temporarily missing for a given city,

[5] Our extractor is implemented using Python [14] version 1.3.

[6] Comparison tests have shown a processing speed of 10K/sec. of text for the native HTML parser vs. 2 MB/sec. for the Python `find` command.

etc.) from which our first prototype did not recover. We are now in the process of extracting other kinds of semistructured information from the Web and find that the currently implemented set of extraction operators is powerful enough to handle all of the encountered sources; i.e., our specification files are straightforward and easy to understand.

References

[1] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," In *Proceedings of Tenth Anniversary Meeting of the Information Processing Society of Japan*, Tokyo, Japan, 7-18, 1994.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel Query Language for Semistructured Data," In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[3] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, A. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, "Towards heterogeneous multimedia information systems: the Garlic approach," In *Proceedings of Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, Los Angeles, California, 123-130, 1995.

[4] T. Kirk, A. Levy, J. Sagiv, and D. Srivastava, "The Information Manifold," AT&T Bell Laboratories, Technical Report 1995.

[5] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, "The RUFUS System: Information Organization for Semi-Structured Data," In *Proceedings of Nineteenth International Conference on Very Large Databases*, Dublin, Ireland, 97-107, 1993.

[6] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman, "A Query Translation Scheme for Rapid Implementation of Wrappers," In *Proceedings of Fourth International Conference on Deductive and Object-Oriented Databases*, Singapore, 1995.

[7] Weather Services International. "INTELLICAST: Europe Weather." URL, http://www.intellicast.com/weather/europe/.

[8] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object Exchange Across Heterogeneous Information Sources," In *Proceedings of Eleventh International Conference on Data Engineering*, Taipei, Taiwan, 251-260, 1995.

[9] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, and R. Yerneni, "Template-Based Wrappers in the TSIMMIS System," In *Proceedings of Twenty-Third ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[10] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, "Object Fusion in Mediator Systems," In *Proceedings of Twentieth International Conference on Very Large Databases*, Bombay, India, 1996.

[11] S. C. Johnson, "Yacc—yet another compiler compiler," AT&T Bell Laboratories, Murray Hill, N.J., Computing Science Technical Report 32, 1975.

[12] N. Ashish and C. Knoblock. "Wrapper Generation for Semi-structured Internet Sources." *Workshop on Management of Semistructured Data*, Ventana Canyon Resort, Tucson, Arizona.

[13] L. Wall and R. L. Schwartz (1992). *Programming perl*, O'Reilly & Associates, Inc., Sebastopol, CA.

[14] Corporation for National Research Initiatives. "The Python Language Home Page." URL, http://www.python.org/, Reston, Virginia.