

A Framework for Generating Attribute Extractors for Web Data Sources

Davi de Castro Reis, Robson Braga Araújo, Altigran S. da Silva, and Berthier A. Ribeiro-Neto

Department of Computer Science
Federal University of Minas Gerais
31270-901 - Belo Horizonte MG - Brazil
{braga,davi,alti,berthier}@dcc.ufmg.br

Abstract. To cope with the irregularities of typical semistructured Web data, extraction tools usually break the extraction task in two phases: an extraction phase, in which atomic attribute values are extracted from Web pages, and an assembling phase, in which these atomic values are grouped to form complex objects. As a consequence, the whole process is highly dependent on the attribute values collected in the first phase. All attribute values of interest should be properly recognized and spurious values should be discarded. Thus, attribute values extraction is an important problem. In this paper, we propose a new framework for generating attribute value extractors. The main appeal of this framework is that it can be adapted for dealing with specific types of data sources and to incorporate distinct types of heuristics for achieving good extraction performance. To demonstrate the feasibility of this proposal, we present an implementation of this framework for data-rich Web pages and show how a number of simple heuristics, some of them presented in the recent literature, can be incorporated into this framework. We also show experimental results and, in most cases, our results are at least as good as results previously presented in the literature.

1 Introduction

In the past few years, the number of sites and services available on the Web that can be regarded as data “containers” has increased steadily. Such large availability has opened the possibility of using these data in a variety of ways. However, *data-rich* or *data-intensive* [6] Web sources most often provide only HTML pages in which data of interest (e.g., data on featuring movies) appears implicit. Its structure can be detected by virtual inspection but has not been declared explicitly. In most cases, such data occur mixed inside Web page text with markup tags, other strings, in-line code, etc. Further, the structure of the data is only suggested by presentation features. Such a structure is often loose, with the possibility that two similar items (e.g., entries on two distinct movies) present structural variations between them. Because of this, data available in Web sources is classified as being *semistructured* [1].

To exemplify, Figure 1 illustrates the excerpt of a Web page, obtained in response to the query “Paul McCartney” submitted to the *Amazon.com* Web site. This page contains data about several distinct products related to the query, organized by sections that correspond to *Amazon.com* stores. To simplify our example, we consider only the stores *books* and *auctions*. Consider we are interested in extracting data on the stores and the products available on it. In the page excerpt of Figure 1, we can identify two *complex objects*: Books and Auctions. Both objects are of type Store. They are composed of an atomic value that identifies the store name and a list of Item objects, each of them being a complex object itself. Note that, for each store, the information on products is distinct. The information can be structured as an aggregation of attributes Item, AuthorList and BookType, for the Books store, and of attributes Item, Bid and Time, for the Auctions store.

A commonly adopted strategy for gaining access to implicit Web data is to build *wrappers* that extract the data of interest, uncover its semantics, and convert them to a suitable format such as relational tables or XML objects. These converted data can then be adequately processed according to specific application needs. Until recently, the most common approach for developing wrappers was to write them directly using general purpose languages such as *Perl* and *Java*.



Fig. 1. An example Web page from Amazon.com.

Developing wrappers manually has many well known shortcomings, mainly due to the difficulty in writing and maintaining them. More recently, several works in the literature have discussed approaches for semi-automatically generating wrappers for Web data [6, 11, 2].

Recent works in the literature propose the semiautomatic generation of wrappers by deriving the extraction rules w from a given set of examples of the objects to be extracted. According to this approach, given a set $E \subset O$ of example objects, taken from a subset $T_0 \subset T$ of a Web source S , a wrapper generation procedure g generates the extraction rules w . That is, $g(E, T_0) = w$.

In general, example-based approaches for generating wrappers use techniques from machine learning [14] or information retrieval [11]. Although one of the first works in this area [10] suggests the use of an “oracle” for providing examples, most of the research later developed rely on humans for this task [11].

To properly deal with the hierarchical and semistructured nature of the data to be extracted, state-of-art approach for Web data extraction divide the process in two distinct phases: an *extraction* phase and an *assembling* phase. In the extraction phase, atomic attribute values (e.g. names of authors, types of books, etc.) are identified and actually extracted from a given source page. In the *assembling* phase, the attribute values extracted in the previous phase are grouped to assemble complex objects. This two-phase approach is adopted by many recent works in the literature [5, 7, 8, 12, 13, 15, 14], because it provides for greater flexibility. The assembling phase is usually guided by a description of the target structure that is either implicitly assumed [8], obtained from a user [11, 13, 14, 15] or automatically inferred [5].

In this work we are particularly interested in the extraction phase, that is, our focus is on the the problem of extracting attribute values that compose complex semistructured objects. Indeed, this phase is critical for the whole extraction process to work properly.

We present an example-based framework for generating *attribute extractors*. Each such extractor can then be used by a wrapper for extracting values of a given attribute. Several works in the literature propose strategies for generating attribute extractors [8, 13, 15]. The main appeal of the framework we propose here is that it provides hooks that allow direct incorporation of several application-oriented heuristics, with no need of modifying the code of the framework itself. Using our framework, wrapper developers can customize the generation of attribute extractors for specific application needs and orient the extraction goals according to these needs.

Moreover, differently from most works in the field, the presented framework is very flexible. Actually, it is immediately ready for applications other than Web data extraction. By properly tuning the customizable components of the framework, called *delimiters tree* and *Guiders*, we can rapidly develop semi-automatic or automatic wrapper generation applications for several data sources, ranging from legacy proprietary database formats to web server log files.

To demonstrate the feasibility of our proposal, we present an implementation of this framework for data-rich Web pages [6] and show how simple heuristics can be incorporated into the framework. We also show experimental results achieved by using these heuristics individually and by combining them. Our results are at least as good as results previously presented in the literature.

The paper is organized as follows. Section 2 presents the notion of attribute extractors. Section 3 discuss PDS, the type of attribute extractors generated by our framework, while in Section 4 we present our strategy for obtaining a useful PDS. In Section 5 we describe how application-oriented heuristics can be incorporated to the framework. Section 6 describes an implementation of the framework targeted at Web pages and Section 7 presents experimental results obtained with this implementation. In Section 8, we briefly comment the related work. Finally, Section 9 presents our conclusions and comments on future work.

2 Attribute Extractors

This section presents the concept of an *Attribute Extractor*. Through the paper we use the term attribute to refer to atomic components of objects, similarly to what is done in databases.

Informally, we call attribute extractor a regular expression, grammar or finite-state automaton aimed at extracting from an input text (e.g., a Web page) values for a given attribute. Consider the HTML page shown in Figure 2. In this simple page, we can see a list of two books represented by their titles along with the names of their authors.

```
<HTML><BODY BGCOLOR=FFFFFF>
<B>The Catcher in The Rye</B> by <I>Sallinger, J.D.</I>
<B>Zen and The Art of Motorcycle Maintenance</B> by <I>Pirsig, R. M.</I>
</BODY></HTML>
```

Fig. 2. An example Web page.

Suppose we are interested in extracting the names of the authors. For this, we build some regular expressions¹ as shown in Figure 3.

<code><I>.*?</I></code>	<code><\w+>.*?</\w+>\n</code>	<code>e by <I>.*?</I></code>
(a)	(b)	(c)
<code>Rye by <I>.*?</I></code>	<code><\w+>.*?</\w+></code>	<code>.*?</code>
(d)	(e)	(f)

Fig. 3. Regular expressions for extracting author names from the Web page of Figure 2.

The regular expressions of Figures 3(a)–(e) can be considered as attribute extractors for author names, since all of them match strings corresponding to the name of an author in the page of Figure 2. The regular expression in Figure 3(f), however, does not match any author name and, thus, it is not an attribute extractor for author names.

There are two important properties a useful attribute extractor should have. First, it should be *sound*, that is, it should only match values in the domain of the attribute considered. Second, it should be *complete*, that is, it should match all values in the domain of the attribute considered, occurring in a given text or page.

To exemplify, the regular expressions of Figure 3(a)–(c) correspond to attribute extractors that are both sound and complete. Figure 3(d) correspond to an attribute extractors that is sound, but not complete, since it only extracts the value *Sallinger, J.D.*, while the regular expression in Figure 3(e) corresponds to an attribute extractor that is complete, but not sound, since it extracts strings that are not author names. These properties are formalized in Definition 1.

¹ We adopt here a usual notation for regular expressions similar to Perl, Grep, etc.

Definition 1. Let τ be an attribute whose domain is D_τ , and let $D_\tau(g)$ be the set of strings in a Web page g that belong to D_τ . Let \mathcal{E}_τ be an attribute extractor for τ and L_ϵ be the language it denotes. We say that \mathcal{E}_τ is **sound** iff $D_\tau(g) \supseteq L_\epsilon$. Also, we say that \mathcal{E}_τ is **complete** iff $D_\tau(g) \subseteq L_\epsilon$.

From our discussion, one can foresee that there are infinitely many possibilities for generating attribute extractors for a given domain, even if we limit ourselves to attribute extractors that are sound and complete.

To cope with such a complexity, a commonly adopted approach is to break the input document into *tokens*. For this, it is necessary to assume some tokenization policy. In Figure 4 we present an excerpt of a possible tokenization² of the page in Figure 2. Notice that we have adopted here a very simple tokenization policy.

... The • Rye by • <I> Sallinger, • J.D. </I> Zen ...

Fig. 4. Excerpt of a possible tokenization of the Web page in Figure 2.

By using tokenization, it is possible to construct attribute extractors that match tokens instead of single characters. Under this assumption, an attribute extractor is described as a sequence of *slots*, each slot matching a token in the input text. As an example, consider the three regular expressions presented in Figure 5. They correspond to attribute extractors for author names in Figure 2.

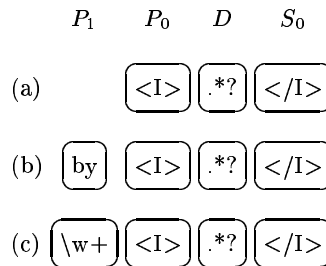


Fig. 5. Examples of attribute extractors.

In the regular expression in Figure 5(a), the slot labeled D matches an author name, and the slots labeled P_0 and S_0 match respectively a token on the left and a token on the right of the author name. In the regular expression in Figures 5(b)–(c), an additional slot (P_1) was used.

Several recent works on Web data extraction adopt concepts similar to attribute extractors based on text tokenization. This is the case of the *AVP-Patterns* used in DEByE [11], the *Finite State Transducers* used in SoftMealy [8], the *Landscape Automata* used in Stalker [14], and the *DataFrames* used by Embley et. al. in their work [6].

In this paper, our main goal is to investigate the design and implementation of a particular type of attribute extractor called *Prefix-Data-Suffix Extractor* or *PDS* extractor for short. The attribute extractors of Figure 5 are all examples of PDS extractors. A PDS extractor is composed of one *data* slot, one or more *prefix* slots (which occur on the left of the data slot), and one or more *suffix* slots (which occur on the right of the data slot). In the next section, PDS extractors are discussed in detail.

² We represent blank spaces by •.

3 Prefix-Data-Suffix Extractors

Let us first introduce the notion of a *delimiters tree*. This notion is central to our ideas. Let $P_m, P_{m-1}, \dots, P_1, P_0, D, S_0, S_1, \dots, S_{n-1}, S_n$ be an attribute extractor for values of an attribute τ occurring in a Web page g . Each P_j or S_i is a slot (i.e., a regular expression) that matches a token in g , and D is any token that occurs between P_0 and S_0 . For this attribute extractor to be useful, its slots must refer specifically to the tokens generated from the target input. Because of this, attribute extractors slots and tokens should be based on a same common tokenization policy. The structure responsible for this policy is what we call a *delimiters tree*.

Definition 2. A *delimiters tree* D is a hierarchy (a tree) in which each node n is associated with a regular expression e_n over an alphabet Σ that denotes a language L_n , such that the following properties apply:

- Let r be the root of D , then $e_r = \Sigma^*$;
- Let r be the root of D and $\{c_1, \dots, c_k\}$ be the set of its children, then $\bigcup_{i=1}^k L_{c_i} = L_r$.
- Let n be a node in D and $\{c_1, \dots, c_k\}$ be the set of its children, then $\bigcup_{i=1}^k L_{c_i} \subseteq L_n$ and $\bigcap_{i=1}^k L_{c_i} = \emptyset$.

The delimiters tree is used with two objectives. First, it guides the tokenization of the input document. The regular expressions in the second level of the delimiters tree are used to extract a sequence of tokens from the target page, assigning higher priority to the bigger tokens. Second, its nodes are used to compose the slots that form attribute extractors.

By building a delimiters tree, it is possible to specialize the framework to generate attribute extractors for a specific class of text documents. Implicitly, by defining such a tree the wrapper developer takes advantage of the structural elements typical of a documents class (e.g. HTML) to allow the construction of suitable attribute extractors. As an example, in Figure 6 we illustrate a delimiters tree built for specializing the framework for HTML documents.

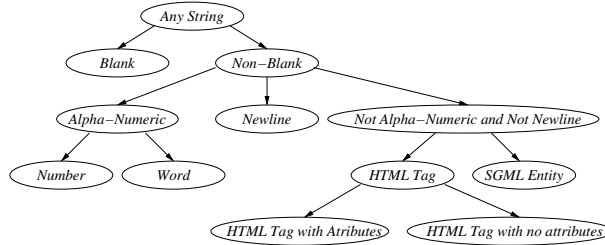


Fig. 6. A delimiters tree for HTML documents.

Attribute extractors generated according to a delimiters tree are termed *Prefix-Data-Suffix Extractors* or *PDS extractors* for short. More formally, we have the following definition:

Definition 3. Let g be a Web page and e be a string in g . Also, let T be a delimiters tree and $g_T = \dots, p_m, \dots, p_0, e, s_0, \dots, s_n, \dots$ be a tokenization of g according to T , where e is a token in g_T . A **Prefix-Data-Suffix Extractor** or **PDS extractor** is an expression of the form: $P_m, P_{m-1}, \dots, P_1, P_0, D, S_0, S_1, \dots, S_{n-1}, S_n$, where D is a regular expression denoting Σ^* , P_i is the i -th token to the left of e in g , or a regular expression ε_v , where v is a node in the delimiters tree T that matches this token, and S_j is the j -th token to the right of e in g , or a regular expression ε_u , where u is a node in the delimiters tree T that matches this token.

We notice that a given PDS extractor π , as any regular expression, denotes a language L_π . From the above definition, the matching portion D of any word belonging to L_π is the data the

PDS extracts. A PDS π is said to be complete if the words in L_π contain all data to be extracted. Similarly, a PDS is said to be sound if all the words in L_π contain only valid data. Therefore, our ultimate purpose is to construct a PDS extractor that is sound and complete. The next session presents the strategy adopted in our framework for obtaining a suitable PDS extractor.

4 PDS Tree

In this section we describe our general strategy to obtain suitable PDS extractors for extracting attribute values from Web pages. As we shall see, this strategy consists in first generating a set of candidate PDS extractors and then using application-oriented heuristics to select from this set the most promising PDS extractor in terms of soundness and completeness.

Let g be a Web page and T be a delimiters tree used to generate a tokenization g_T of g . Given a token e , which is an *example* of a value of an attribute, say τ , it is possible to determine the set of all possible PDS extractors that match this token by using the regular expressions in T . For this, the tokens surrounding the example are turned into slots and combined in all possible ways. The set of these PDS extractors is called the *PDS candidate set*.

As the delimiters tree T defines an hierarchy over these regular expressions, it is also possible to define a hierarchy over the candidate PDS set. This results in a structure we call a *Candidate PDS Tree* or *PDS Tree*, for short.

Definition 4. Let g be a Web page, T be a delimiters tree and $g_T = \dots, p_m, \dots, p_0, e, s_0, \dots, s_n, \dots$ be a tokenization of g according to T . We define a **PDS Tree** in g as follows:

- Each node in the tree is a PDS extractor of the form $P_m, \dots, P_0, D, S_0, \dots, S_n$, where D is a regular expression denoting σ^* and $P_i (S_j)$ is either equal to $p_i (s_j)$ or it is a regular expression in T that matches $p_i (s_j)$;
- The root of the tree is the PDS p_0, D, s_0 ;
- If $P_m, \dots, P_0, D, S_0, \dots, S_n$ is a node in the tree, then
 - every PDS $P_{m+1}, P_m, \dots, P_0, D, S_0, \dots, S_n$ is a child of this node in the tree;
 - every PDS $P_m, \dots, P_0, D, S_0, \dots, S_n, S_{n+1}$ is a child of this node in the tree;

Informally, a PDS Tree is generated for a token e , given as *example*, such that all PDS extractors in the tree are capable of extracting e and, possibly, other strings. The children of a given PDS extractor in a tree are those that have one more slot (to the left or to the right), being this slot the token (to the left or to the right) itself, or some equivalent regular expression obtained from the delimiters tree.

As an example, consider the example page of Figure 2, the delimiters tree of Figure 6, the tokenization of Figure 4 and the PDS extractor of Figure 5. In Figure 7 we illustrate an excerpt of a PDS Tree generated when the string Sallinger, J.D. is given as example. In the root of this subtree is the PDS labeled (a) formed using the tokens that occur immediately on the left and on the right of the string provided as example. Among the children of this PDS we find the PDS labeled (b), which adds a token to the left, and the PDS labeled (c), in which the token on the left (by) is replaced by a regular expression ($\langle \backslash w + \rangle$) taken from the delimiters tree of Figure 6.

Our next step consists in traversing the PDS Tree in search of a suitable PDS. A naive strategy to accomplish this would be to exhaustively traverse the whole tree, apply each PDS to the target page and use an oracle (e.g., a human user) to determine whether the PDS was able to correctly extract all values of interest and only those values. This strategy is not feasible for two main reasons. First, its not possible to guarantee that a sound and complete PDS exists, and, second, the complete traversal of the whole PDS Tree is unbearable in practice.

The tree-like organization of the structure comes in hand to solve this problem. We can see PDS tree as a decision tree, where each node is presented to the user as a possible solution. If the user is not satisfied with such solution, then he points the node that seems to be nearer to the optimal solution. As user feedback is a high cost resource, we will try to simulate it by means of decision making functions that encapsulate application-oriented heuristics. This functions are called *Guiders* and are discussed in the next section.

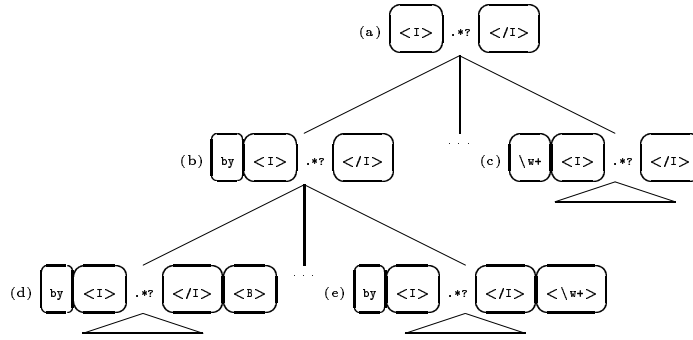


Fig. 7. Example of a PDS Tree

5 Guiders

Given a PDS tree as described above, consider an oracle³ that is capable of telling for a given node n reached during the traversal: (1) whether the traversal must stop, meaning that n is “reasonable” PDS; or (2) which child of n is the “most promising” node for the traversal to continue. Such a traversal is describe by the PDS-Search algorithm in Figure 8.

```

1 PDS-Search( $n$  : root of a subtree of a PDS Tree);
2
3 begin
4   if Oracle.Stop( $n$ )
5     then Return  $n$  ;
6   else  $c \leftarrow$  Oracle.ChooseChild( $n$ ) ;
7         PDS-Search( $c$ ) ;
9   fi
10 end

```

Fig. 8. The PDS-Search Algorithm.

The Oracle described above is the second main component of the framework we propose in this paper. It is used to encapsulate a number of heuristics that can provide the two kinds of information required by the PDS-Search algorithm. In our framework, these heuristics are implemented by small functions called *Guiders*, since they have the role of guiding the traversal on the PDS Tree in the search for a “reasonable” PDS.

Guiders are invoked by the Oracle for each node traversed according to the PDS-Search algorithm. As a response, each guider must be capable of returning two values called *vote* and the *stop*. The *vote* value is used to choose one of the children of the current PDS as a next step. The *stop* value is used to know if this PDS is considered good enough so that the tree traversal can stop. Besides the *vote* and the *stop* value, guiders must return a *confidence level* associated to each of these values.

For each node n found in the traversal, the oracle must decide whether it must go on the traversal or not. If it takes the decision of going on, it must choose one of the children PDS of n . To make this decision, the oracle invokes all guiders available to evaluate each of the children. For each confidence level returned by the guiders, an average of the vote values for each child PDS is computed. Then, the oracle selects the child that received the highest vote average within the highest confidence level. In the case of a draw, the child that received the highest vote average within the second highest confidence level is selected and so on. The decision of stopping the

³ This oracle is not related to oracles that can be used for automatically providing examples

traversal in node n is taken in a similar fashion. However, this is only taken if the highest average of stop values within the highest confidence level is greater than a predefined threshold. This is a very simple approach to combining evidences given by the guiders, but it showed to have enough power for our purposes.

Our framework allows guiders to have access to all information available during the process. That is, guiders have access to the example given, to the delimiters tree, to the target page and its tokenization, and, of course, to the PDS Tree. In addition, guiders may also receive parameters. Thus, guider developers are allowed to use any of these structures to compute votes and to establish their confidence levels. For instance, guiders can take the current PDS in the traversal, use it to extract from the input page and then cast its vote and confidence level based on the results of the extraction.

Guiders constitute, along with delimiters tree, the main customization resources within our framework. By properly building them, wrapper developers can introduce heuristics for dealing with specific type of text or with special kinds of data of their interest. They specify the biases of the wrapper induction system.

By making guiders properly declaring their confidence level, developers can prevent a guider that is too specific from compromising more generic guiders. For instance, a guider built to operate inside columns of an HTML table can declare a low confidence level when invoked outside an HTML table, so its vote is deprecated in favor of “more confident” guiders. Notice that we allow wrapper developers to choose only the guiders needed during a wrapper generation session, that is, it is not necessary to use all guiders available at once.

6 Putting all together

To demonstrate the feasibility of the proposed framework, we created an implementation of it targeted to HTML documents. First, we built a delimiters tree similar to that presented at Figure 6 for exploiting the typical structural features of HTML documents. Next, we also created a few guiders to guide the traversal of the PDS Tree. Following, we briefly describe the guiders implemented.

- **Position Guider.** To evaluate a PDS, this guider bases its decision on the relative positions of the atomic values extracted by the PDS. For this, the guider uses the PDS to extract from the input page and verifies the standard deviation of the distances (in the page) between the atomic values obtained. PDS that result in small values for this metric are better evaluated. The heuristic implemented by this guider is similar to one of the heuristics used in [6].
- **Counter Guider.** This is an example of guider that receives a parameter. In this case, the guider takes from the user an estimate of the number of values of the attribute of interest in the input page. PDS extractors that extract nearly this number of atomic values are better evaluated. This guider emulates the heuristic use in [11].
- **HTML Tree Guider.** This guider takes advantage of the inherent structure of the HTML parse tree corresponding to the input page. When evaluating a given PDS, this guider determines, for each atomic value obtained by the PDS, the path in the parse where it is located. PDS that result in atomic values with similar paths are favored when evaluated by this guider. This guider provides a functionality similar to that of [4].
- **MSYFM Guider.** MSYFM (Make Sure You Find Me) is a guider that takes as parameters additional examples of values of the attribute of interest. This guider tries to ensure that all examples given are covered by the chosen PDS.

Concerning MSYFM Guider, it is worth noticing that some approaches in the literature [11] prefer to create distinct attribute extractor for different partitions of the set of attribute values and combine these attribute extractor using disjunctions. In our case, this can be easily accomplished by an application (i.e., a wrapper induction system) that uses our framework. The application can verify the failure of the MSYFM Guider, try new examples and create distinct PDS extractors,

simulating the disjunction functionality. This is not an inherent part of the framework since we also support imperfect oracles [7] for providing examples.

In Figure 9 we present a diagram that illustrates the implemented framework. In summary, to generate a PDS for an attribute of interest, the framework takes an input page and one Main Example provided by the user. The Tokenizer then takes the Input Page and generates the Tokenization based on the Delimiters Tree. Next, the PDS Tree Builder takes the Main Example, the Tokenization and the Delimiters Tree and generates the PDS Tree. The PDS Tree is traversed by the Oracle relying on the decisions taken by the guiders. We recall that, although not illustrated in Figure 9, the guiders have access to all structures manipulated by the framework during the process.

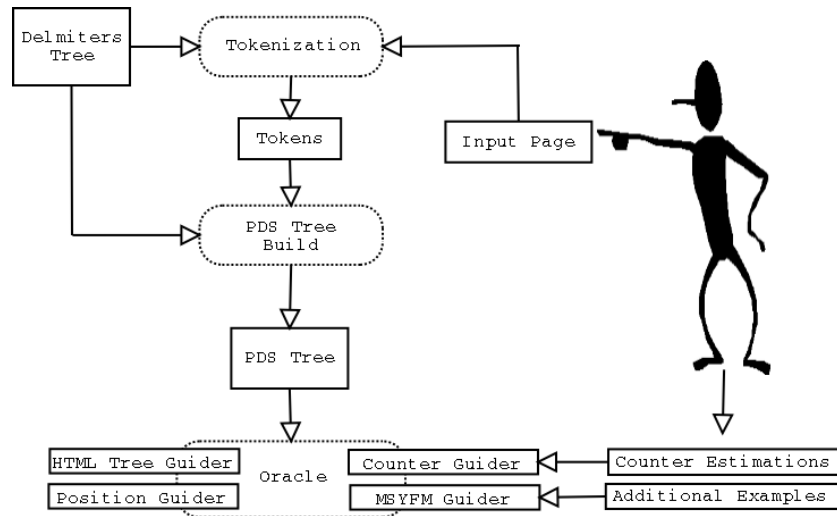


Fig. 9. Whole framework in action

7 Experimental Results

This section presents results from experiments we have carried out with the implemented framework described in Section 6. The experiments consist in generating sound and complete PDS extractors for extracting values of attributes of interest from typical data-rich Web pages. These Web pages were taken from several distinct Web sites, some of them previously used for experiments in other works in the literature [4, 11, 10].

Three guiders were used in the experiments: MSYFM, HTML Tree and Position. We do not use the Counter Guider, since we wanted to limit the role of users to simply providing examples. As for the remaining guiders, all extractions used the MSYFM Guider, and, for each site, we first tried the HTML Tree Guider and the Position Guider separately and then tried both guiders together. For each attribute, we generate a single PDS. Notice that none of the guiders were specially tuned for a given page, and user feedback was not used.

The obtained results are shown in Table 1. In this table, column Ex. indicates the maximum number of examples used for each attribute. Column Sound shows whether the best final PDS extractor obtained among those generated is sound or not. The same applies to Complete column. Column Tree indicates whether the HTML Tree Guider individually was able to reach such result. The same applies to column Pos with respect to the Position Guider and to column Both with respect to the combined oracle of both guiders.

As Table 1 shows, in most of the 55 experiments our framework was able to generate PDS extractors that are sound and complete. In 4 cases (Attributes 6, 41, 45 and 46) the PDS extractors

Source	#	Attribute	Ex.	Tree	Pos	Both	Sound	Complete
Bigbook	1	Address	1	x	x	x	x	x
	2	Business	2			x	x	x
	3	City	1	x		x	x	x
	4	Tel	1	x	x	x	x	x
ComputerESP	5	Price	1	x	x	x	x	x
	6	Product	2					
	7	Company	1	x			x	x
Altavista	8	Title	1	x	x	x	x	x
	9	Description	2	x	x	x	x	x
	10	URL	2	x	x	x	x	x
	11	Size	1	x	x	x	x	x
	12	Date	1	x	x		x	x
13	Language	2	x	x	x		x	
CineMachine	14	Movie Info	1		x	x	x	x
News.COM	15	Title	1		x		x	x
	16	Date	1	x		x	x	x
SiteSeeker	17	Site	1		x	x	x	x
	18	Description	1		x	x		x
Job Newsgroups	19	Headline	2	x			x	x
	20	Score	2		x	x	x	x
New Journal	21	Title	2	x				x
JOBS jobs Jobs	22	Job	2	x	x	x	x	x
	23	Date	1	x			x	x
Amazon - Cars	24	Car	2		x	x	x	x
	25	Price 1	1	x	x	x	x	x
	26	Price 2	1		x		x	x
	27	Company	1		x		x	x
Barnes E-books	28	Title	1		x	x	x	x
	29	Genre	1	x		x	x	x
	30	Price	1	x	x	x	x	x
	31	Author	1			x	x	x
	32	Description	1		x		x	x
Buy.com	33	Product	1		x	x	x	x
	34	Price	1		x		x	x
	35	Platform	1		x	x	x	x
	36	Midia	1		x	x	x	x
	37	Manufacturer	1	x	x	x	x	x
Travelocity	38	Zone	1	x				x
	39	Hotel	1		x		x	x
	40	Price	1	x			x	x
	41	Nights	1					
Restaurant	42	Restaurant	1	x			x	x
	43	Address	1	x			x	x
	44	Description	1	x	x		x	x
	45	Credit Card	1					
	46	Telephone	1					
Slashdot	47	Title	1	x	x	x	x	x
	48	Text	2		x		x	x
	49	Date	1	x	x	x	x	x
	50	Author	2	x	x	x	x	x
Freshmeat	51	Application	1		x		x	x
	52	About	1		x		x	x
	53	Changes	1		x	x	x	x
	54	Author	2	x	x	x	x	x
	55	Date	1		x		x	

Table 1. Results of the experiments

were neither sound nor complete. In 4 cases (Attributes 13, 18, 21 and 38) they were just complete and in 1 case (Attribute 55) the obtained PDS extractor were just sound.

In most cases for which the generated PDS extractor were just complete, the values extracted incorrectly could be easily cleaned by using techniques like *Corroboration* [10] or ignored during the assembling of complex objects by using a technique like *Hot Cycles* [5]. The *date* attribute of *Freshmeat* was just sound, because there were some dates with a little different local context. A disjunction led us to the correct result.

The first attribute extraction that failed completely (Attribute 6), failed because the generated PDS tree had no sound or complete PDS extractor for it. When we properly extended the delimiters tree used, sound and complete PDS extractors were generated in the PDS tree and guiders were able to find it. For the case of Attribute 45, by using three distinct PDS extractors combined by disjunction, we successfully extracted the attribute values. For the two remaining cases (Attributes 41 and 46), the context was ambiguous and could not be described by a regular grammar such as the one used by PDS extractors.

8 Related Work

In the recent literature, a number of works on the generation of wrappers for Web data sources have been presented. A brief survey of these works can be found in [12].

Kushmerick et. al. [10] were the first to present formally the wrapper induction problem. Since then, several works [4, 11] tried to present specific wrapper induction systems for semistructured Web data extraction. Finally, [3] proposed an extensible architecture for creating wrapper induction systems.

SoftMealy [8], by Hsu et. al., uses *finite-state transducers (FST)* for learning extractors. Their induction technique does not directly compare to our work, since we only deal with single attributes. Contextual rules, that constitute the edges connecting the FST states, are analogous to our PDS. SoftMealy relies on set-covering techniques to induce such rules. Our approach, based on guiders, is more flexible, still allowing to use the same techniques. SoftMealy is, however, more powerful in some cases. This is because contextual rules are reinforced by the current state information provided by the FST itself. To achieve the same level of expressiveness, we would need to extend the PDS extraction language to allow loops at a given slot, and null slots. With this change, our framework would provide a functionality closer to SoftMealy.

RoadRunner [4], by Crescenzi et. al., is a wrapper generation system targeted to HTML pages. It also has a fixed extraction language and fixed wrapper induction techniques. The main appeal of this tool is that it is able to automatically generate a Wrapper based only on a set of given sample pages. No user-provided examples are needed. On the contrary, our framework is based on examples. Indeed, relying on examples is considered safer in many situations [12]. However, we could use an automatic tool to provide the examples, such as [7]. Once more, our PDS extractor cannot cover all cases where is possible to generate a RoadRunner extractor. The same extensions to the language needed to reach SoftMealy expressiveness could be used to equalize both tools. The wrapper induction techniques used in RoadRunner could be reproduced by a guider taking as parameter additional sample pages. The HTML tree guider we have implemented introduces a few of the RoadRunner biases.

Stalker [14], by Muslea et. al., is a wrapper induction systems that uses *landscape automata*, a type of attribute extractor similar to PDS. It has a very powerful extraction language, and a fixed extractor inducing approach. Different from most works, Stalker heavily relies on disjunctions, that is, distinct extractors to deal with different partitions of the set of an attribute values in the target page. While it would be hard to change our framework to cover all extractors that can be created by Stalker, the same modifications proposed above could take PDS very near to Stalker's extraction language. Our framework do not support disjunction directly. In order to emulate Stalker bias, the application implementing the framework should introduce a guider that identifies the need of a disjunction, and then give the examples belonging to each partition separately to the framework, restarting the extractor inducing cycle.

To the best of our knowledge, from the works in the literature, the Structured Wrapper Induction System [3] by Cohen et. al. is the one that most resembles our framework in terms of flexibility. Cohen relies on a single learning system and builders to define the bias of the system. Builders are analogous to our guiders, but they define extraction languages and are combined through specially hand-crafted functions, complying to a pre-defined interface. Our framework has a fixed extraction language, and guiders are combined either automatically, or through confidence levels definitions. This system can be tuned to work at very different targets, but requires a high effort and some expertise in the field in order to do so. We try to make guider construction a very simple task, and provides the possibility of applications automatically tuning their combination. While the Structured Wrapper Induction System is targeted at experts providing fast wrapper solutions, our framework is targeted at developers of wrapper induction systems for non-expert users.

From the above comparisons, we could see that the inherent extraction language of PDS is not the most powerful available. Its advantage is that it permits simple and efficient learning. More complex languages requires more complex learning systems. We chose to create good wrappers for a limited language. Further, the experimental results showed that a properly constructed PDS extractor is as good as any other extractor.

9 Conclusions and Future Work

We have presented an extensible framework for generating attribute extractors. The main appeal of this framework is that it can be adapted for dealing with specific types of data sources and to incorporate distinct types of heuristics for achieving good extraction performance. We have also demonstrated the feasibility of this proposal by implementing the framework along with a number of guiders that encapsulate simple heuristics to deal with Web pages. Experimental results we have presented demonstrate the effectiveness of the implemented framework with several typical Web data sources.

We recall that the main goal of our framework is to generate attribute extractors to be used by wrappers in the the extraction of atomic attribute values. The problem of grouping these values to compose complex objects is not addressed by our framework. It is the main subject of works such as [5] and [9].

In comparison to other wrapper generation systems, our framework can be considered one that requires *low effort for customization*. Empirical observation showed that: (1) the underlying PDS extraction language is expressive enough to address the majority of extractors needed; (2) guiders are able to introduce most of the biases or heuristics wrapper induction systems needs. Our framework takes these observations into account, and let as the only tasks for developers creating a delimiters tree that exploits the target document class structure and writing appropriate guiders to produce high quality extractors. The experimental results showed that a very simple implementation of the framework, with no special tuning, disjunction or user feedback was able to produce results comparable to those found in the related works, with very few examples.

The performance of the framework was not the main concern for the reference implementation, but still we achieved good results. The extractions sessions usually occur in small pages, and takes no more than a second in a standard desktop machine. Extraction from large repositories usually takes longer because of the cost of the tokenization process. For instance, extracting the City from the Bigbook Web source from a 3 MB page set took about 31 seconds. This can be considered as satisfactory, since the tokenization takes place just once for several attribute extractions. This result could be prohibitive for online systems, but they usually deal with small repositories. All processes in the framework have linear cost.

As future work, we intend to build a full-featured application that dynamically customizes the framework, and also let the user make manual customizations. Even an automatically delimiters tree generation seems possible, and we will try to accomplish this task. We intend to use these results in the next version of the DEByE Tool [11].

Finally, to verify the applicability of our framework for different document classes, we intend to build a delimiters tree and a set of guiders for extracting information from Postscript files. The

main application would be to extract bibliographical information, abstracts and section titles from scientific articles available in this format.

References

- [1] ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, 1999.
- [2] BAUMGARTNER, R., FLESCA, S., AND GOTTLOB, G. Visual web information extraction with lixto. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)* (Rome, Italy, 2001), pp. 119–128.
- [3] COHEN, W. W., AND JENSEN, L. S. A structured wrapper induction system for extracting information from semi-structured documents. In *Proceedings of the IJCAI-2001 Workshop on Adaptive Text Extraction and Mining* (Seattle, Washington, 2001).
- [4] CRESCENZI, V., MECCA, G., AND MERIALDO, P. RoadRunner: Towards automatic data extraction from large Web sites. In *Proceedings of the 26th International Conference on Very Large Data Bases* (Rome, Italy, 2001), pp. 109–118.
- [5] DA SILVA, A. S. *Example-based Strategies for Extracting Semistructured Web Data*. PhD thesis, Department of Computer Science, Federal University of Minas Gerais, 2002.
- [6] EMBLEY, D. W., CAMPBELL, D. M., JIANG, Y. S., LIDDLE, S. W., KAI NG, Y., QUASS, D., AND SMITH, R. D. Conceptual-model-based data extraction from multiple-record Web pages. *Data and Knowledge Engineering* 31, 3 (1999), 227–251.
- [7] GOLGHER, P. B., DA SILVA, A. S., LAENDER, A. H. F., AND RIBEIRO-NETO, B. A. Bootstrapping for Example-Based Data Extraction. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management* (Atlanta, GA, 2001), pp. 371–378.
- [8] HSU, C.-N., AND CHANG, C.-C. Finite-state transducers for semi-structured text mining. In *Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications* (Stockholm, Sweden, 1999), pp. 38–49.
- [9] JENSEN, L. S., AND COHEN, W. W. Grouping extracted fields. In *Proceedings of the IJCAI-2001 Workshop on Adaptive Text Extraction and Mining* (Seattle, Washington, 2001).
- [10] KUSHMERICK, N., WELD, D. S., AND DOORENBOS, R. Wrapper Induction for Information Extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (Osaka, Japan, 1997), pp. 729–737.
- [11] LAENDER, A. H. F., RIBEIRO-NETO, B., AND DA SILVA, A. S. DEByE – Data Extraction by Example. *Data and Knowledge Engineering* 40, 2 (2002), 121–154.
- [12] LAENDER, A. H. F., RIBEIRO-NETO, B., DA SILVA, A. S., AND TEIXEIRA., J. S. A Brief Survey of Web Data Extraction Tools. *SIGMOD Record* (2002). To appear.
- [13] MUSLEA, I., MINTON, S., AND KNOBLOCK, C. An Hierarchical Approach to Wrapper Induction. In *Proceedings of the 3rd Annual Conference on Autonomous Agents* (Seattle, WA, 1999), pp. 190–197.
- [14] MUSLEA, I., MINTON, S., AND KNOBLOCK, C. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems* 4, 1/2 (2001), 93–114.
- [15] RIBEIRO-NETO, B., LAENDER, A. H. F., AND DA SILVA, A. S. Extracting semi-structured data through examples. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management* (Kansas City, MO, 1999), pp. 94–101.