RoadRunner: Towards Automatic Data Extraction from Large Web Sites

Valter Crescenzi
Università di Roma Tre
crescenz@dia.uniroma3.it

Giansalvatore Mecca
Università della Basilicata
mecca@unibas.it

Paolo Merialdo
Università di Roma Tre
merialdo@dia.uniroma3.it

Abstract

The paper investigates techniques for extracting data from HTML sites through the use of automatically generated wrappers. To automate the wrapper generation and the data extraction process, the paper develops a novel technique to compare HTML pages and generate a wrapper based on their similarities and differences. Experimental results on real-life data-intensive Web sites confirm the feasibility of the approach.

1 Introduction

The amount of information that is currently available on the net in HTML format grows at a very fast pace, so that we may consider the Web as the largest "knowledge base" ever developed and made available to the public. However HTML sites are in some sense modern legacy systems, since such a large body of data cannot be easily accessed and manipulated. The reason is that Web data sources are intended to be browsed by humans, and not computed over by applications. XML, which was introduced to overcome some of the limitations of HTML, has been so far of little help in this respect. As a consequence, extracting data from Web pages and making it available to computer applications remains a complex and relevant task.

Data extraction from HTML is usually performed by software modules called *wrappers*. Early approaches to wrapping Web sites were based on manual techniques [2, 9, 17, 4, 11]. A key problem with manually coded wrappers is that writing them is usually a difficult and labor intensive task, and that by their nature wrappers tend to be brittle and difficult to maintain.

This paper develops a novel approach to the data extraction problem: our goal is that of fully automating the wrapper generation process, in such a way that it does not rely on any a priori knowledge about the target pages and their contents. From this attempt come the main elements of originality with respect to other works in the field, as discussed in the following section.

2 Overview and Contributions

Target of this research are the so-called data-intensive Web sites, i.e., HTML-based sites with large amounts of data and a fairly regular structure. Generating a wrapper for a set of HTML pages corresponds to inferring a grammar for the HTML code – usually a regular grammar – and then use this grammar to parse the page and extract pieces of data. Grammar inference is a well known and extensively studied problem (for a survey of the literature see, for example, [15]). However, regular grammar inference is a hard problem: first, it is known from Gold's works [6] that regular grammars cannot be correctly identified from positive examples alone; also, even in the presence of both positive and negative examples, there exists no efficient learning algorithm for identifying the minimum state deterministic finite state automaton that is consistent with an arbitrary set of examples [7]. As a consequence, the large body of research that originated from Gold's seminal works has concentrated on the development of efficient algorithms that work in the presence of additional information (typically a set of labeled examples or a knowledgeable teacher's responses to queries posed by the learner).

The fact that regular expressions cannot be learned from positive examples alone, and the high complexity of the learning even in the presence of additional information have limited the applicability of the traditional grammar inference techniques to Web sites, and have recently motivated a number of pragmatical approaches to wrapper generation for HTML

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

pages. These works have attacked the wrapper generation problem under various perspectives, going from machine–learning [18, 12, 10, 14] to data mining [1, 16] and conceptual modeling [5]. Although these proposals differ in the formalisms used for wrapper specification, they share a number of common features:

1. First, the wrapper generator works by using additional information, typically a set of labeled samples provided by the user or by some other external tool; the wrapper is inferred by looking at these positive examples and trying to generalize them.

2. Second, it is usually assumed that the wrapper induction system has some *a priori* knowledge about the page organization, i.e., about the schema of data in the page; most works assume that the target pages contain a collection of flat records; in other cases [1] the system may also handle nested data, but it needs to know what are the attributes to extract and how they are nested.

3. Finally, these systems generate a wrapper by examining one HTML page at a time.

This paper investigates the wrapper generation problem under a new perspective. In particular, we aim at automating the wrapper generation process to a larger extent and our approach clearly departs from the ones in the literature in several respects:

1. Our system does not rely on user-specified examples, and does not require any interaction with the user during the wrapper generation process; this means that wrappers are generated and data are extracted in a completely automatic way;

2. The wrapper generator has no a priori knowledge about the page contents, i.e., it does not know the schema according to which data are organized in the HTML pages: this schema will be inferred along with the wrapper; moreover, our system is not restricted to flat records, but can handle arbitrarily nested structures;

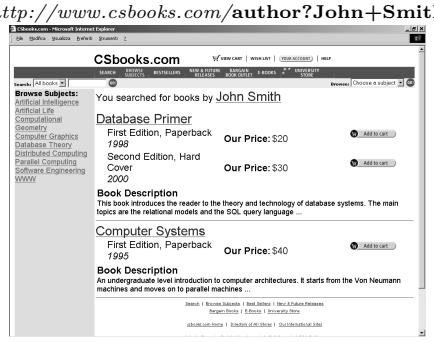
3. We propose a novel approach to wrapper inference for HTML pages; in order to tell meaningful patterns from meaningless ones, our system works with two HTML pages at a time; pattern discovery is based on the study of similarities and dissimilarities between the pages; mismatches are used to identify relevant structures.

2.1 Data Extraction in RoadRunner

Pages in data-intensive sites are usually automatically generated: data are stored in a back-end DBMS, and HTML pages are produced using scripts – i.e., programs – from the content of the database. To give a simple but fairly faithful abstraction of the semantics of such scripts, we can consider the page-generation process as the result of two separated activities: (*i*) first, the execution of a number of queries on the underlying database to generate a *source dataset*, i.e. a set of tuples of a possibly nested type that will be published in the site pages; (*ii*) second, the serialization of the source dataset into HTML code to produce the actual pages, possibly introducing URLs links, and other material like banners or images. We call a *class of pages* in a site a collection of pages that are generated by the same script.

We may formulate the problem studied in this paper as follows: "given a set of sample HTML pages belonging to the same class, find the nested type of the source dataset and extract the source dataset from which the pages have been generated." These ideas are clarified in Figures 1 and 2, which refers to a fictional bookstore site. In that example, pages listing all books by one author are generated by a script; the script first queries the database to produce a nested dataset in which each tuple contains data about one author, her/his list of books, and for each book the list of editions; then, the script serializes the resulting tuples into HTML pages (Figure 1). When run on these pages, our system will compare the HTML codes of the two pages, infer a common structure and a wrapper, and use that to extract the source dataset. Figure 2 shows the actual output of the system after it is run on the two HTML pages in the example. The dataset extracted is produced in HTML format. As an alternative, it could be stored in a database.

*http://www.csbooks.com/***author?John+Smith**



*http://www.csbooks.com/***author?Paul+Jones**

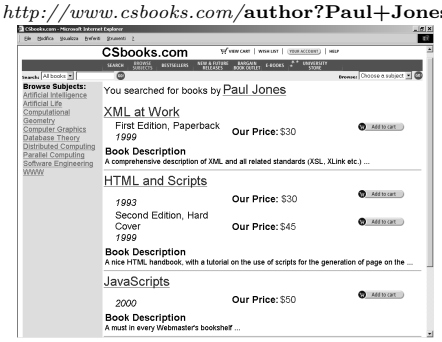


Figure 1: Input HTML Pages

As it can be seen from the figure, the system infers a nested schema from the pages. Since the original database field names are generally not encoded in the pages and this schema is based purely on the content

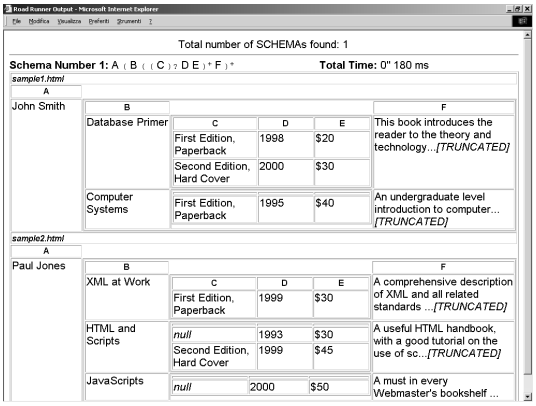


Figure 2: Data Extraction Output

of the HTML code, it has anonymous fields (labeled by A, B, C, D, etc. in our example), which must be named manually after the dataset has been extracted; one intriguing alternative is to try some form of post-processing of the wrapper to automatically discover attribute names. However, this is a separate research problem; for space and simplicity reasons we don't deal with it in this work.

3 Theoretical Background

The theoretical roots of this study can be traced back to a previous paper [8]; in the present work, we build on that theoretical setting, and concentrate on the development of actual algorithms for wrapper generation and data extraction from large HTML sites. The main intuition behind our work is that the site generation process can be seen as an *encoding* of the original database content into strings of HTML code; as a consequence, we see data extraction as a *decoding* process.

We may summarize the theoretical grounds established in [8] as follows. First, we concentrate on *nested types*, built by arbitrarily nesting list and tuple constructors; due to the fact that data items in HTML strings are inherently ordered, in the following we will blur the distinction between ordered and unordered collections. Null values are allowed in type instances. In [8], the schema finding and data extraction problem presented above was formalized as the problem of rediscovering the nested type of a collection of instances encoded as strings for storage purposes, and decoding the original instances from their encoded versions.

To solve the problem, a lattice–theoretic approach is proposed. This is based on a close correspondence between nested types and *union–free regular expressions*. Given a special symbol `#PCDATA`, and an alphabet of symbols Σ not containing `#PCDATA`, a *union-free regular expression (UFRE)* over Σ is a string over alphabet $\Sigma \cup \{\texttt{\#PCDATA}, \cdot, +, ?, (,)\}$ defined as follows. First, the empty string, ϵ and all elements of $\Sigma \cup \{\texttt{\#PCDATA}\}$ are union-free regular expressions. If a and b are UFRE, then $a \cdot b$, $(a)^+$, and $(a)?$ are UFRE. The semantics of these expressions is defined as usual, $+$ being an iterator and $(a)?$ being a shortcut for $(a|\epsilon)$ (denotes optional patterns). We shall also use $(a)^*$ as a shortcut for $((a)^+)?$. The class of union-free regular expressions fits well to this framework since it has a straightforward mapping to nested types (`#PCDATA` map to string fields, `+` map to lists, possibly nested, `?` map to nullable fields). In the following, with an abuse of notation, we will use `(A, B, C, ...)`$^+$ to refer to a type which is a (non empty) list of tuples of the form `(A:string, B:string, C:string, ...)`; `(A, B, C, ...)`* will refer to lists that may be empty. We show in [8] that, given a UFRE σ, the corresponding nested type, $\tau = type(\sigma)$ can be reconstructed in linear time.

Note that, although nested types and UFREs are far from catching the full diversity of structures present in HTML pages, they have been shown [3] to be a promising abstraction for describing the structure of pages in fairly regular Web sites. One obvious alternative could be that of enriching the type system and the grammar formalism by introducing disjunctions, i.e., union operators. However, as discussed above, this would strongly increase the complexity of the wrapper inference process.

Based on the correspondence between nested types and UFREs, it is possible to show that, given a set of HTML strings $s_1, s_2, \ldots s_k$, corresponding to encodings of a source dataset, i.e., of a collection of instances $i_1, i_2, \ldots i_k$ of a nested type τ, we can discover the type τ by inferring the minimal union-free regular expression σ whose language, $L(\sigma)$, contains the strings $s_1, s_2, \ldots s_k$, and then taking $\tau = type(\sigma)$. Also, we can use σ as a wrapper to parse $s_1, s_2, \ldots s_k$ and reconstruct the source dataset $i_1, i_2, \ldots i_k$. Therefore, solving the schema finding and data extraction process amounts to finding the minimal UFRE (if this exists) whose language contains the input HTML strings, $s_1, s_2, \ldots s_k$. If we consider a lattice of UFRE with a containment relationship, such that $\sigma_1 \leq \sigma_2$ iff $L(\sigma_1) \subseteq L(\sigma_2)$, then the UFRE we look for is the least upper bound of the input strings, i.e., $\sigma = \text{LUB}(s_1, s_2, \ldots s_k)$. Since it is known that operator LUB is associative, this in turn amounts to computing the least upper bound of UFREs σ_1 and σ_2, $\text{LUB}(\sigma_1, \sigma_2)$.

Based on the these ideas, the overall schema finding and data extraction process can be solved by iteratively computing least upper bounds on the RE lattice to generate a common wrapper for the input HTML pages. It should be apparent, at this point, that a crucial problem in solving the data extraction problem consists in finding algorithms for computing the least upper bound of two UFREs. In this paper, we concentrate on this problem, and develop an algorithm $match(\sigma_1, \sigma_2)$ to compute the least upper bound of UFREs σ_1 and σ_2, as described in the next section. A

number of experiments on real-life data-intensive Web sites show the effectiveness of the proposed approach, as discussed in Section 5.

4 The Matching Technique

This section is devoted to the presentation of algorithm *match*. It is based on a matching technique called *ACME*, for *Align, Collapse under Mismatch, and Extract*, which we describe in the following.

To avoid errors and missing tags in the sources, we assume that the HTML code complies to the *XHTML* specification, a restrictive variant of HTML in which tags are required to be properly closed and nested (there is no significant loss of generality in this hypothesis, since several tools are available to turn an HTML page into an XHTML one). We also assume that sources have been pre-processed by a lexical analyzer to transform them into lists of tokens; each token is either an HTML tag or a string value (see Figure 3, in which the two HTML samples have been transformed into lists of 20 and 27 tokens, respectively).

The matching algorithm works on two objects at a time: (*i*) a list of tokens, called the *sample*, and (*ii*) a wrapper, i.e., one union-free regular expression. Given two HTML pages (called page 1 and page 2), to start we take one of the two, for example page 1, as an initial version of the wrapper; then, the wrapper is progressively refined trying to find a common regular expression for the two pages. This is done by solving *mismatches* between the wrapper and the sample.

The matching algorithm consists in *parsing* the sample using the wrapper. A mismatch happens when some token in the sample does not comply to the grammar specified by the wrapper. Mismatches are very important, since they help to discover essential information about the wrapper. Whenever one mismatch is found, we try to solve the mismatch by generalizing the wrapper. The algorithm succeeds if a common wrapper can be generated by solving all mismatches encountered during the parsing.

4.1 Mismatches

There are essentially two kinds of mismatches that can be generated during the parsing: (*a*) *String mismatches*, i.e., mismatches that happen when different strings occur in corresponding positions of the wrapper and sample. (*b*) *Tag mismatches*, i.e., mismatches between different tags on the wrapper and the sample, or between one tag and one string. In the following paragraphs we discuss how mismatches can be solved, with the help of the simple example in Figure 3. At the end of this section we will generalize the technique and show how it can be applied to more complex examples that are closer to real sites.

String Mismatches: Discovering Fields It can be seen that, if the two pages belong to the same class, string mismatches may be due only to different values of a database field. Therefore, these mismatches are used to discover fields (i.e., `#PCDATA`). Figure 3 shows several examples of string mismatches during the first steps of the parsing. Consider, for example, strings `'John Smith'` and `'Paul Jones'` at token 4. To solve this string mismatch, we simply need to generalize the wrapper to mark the newly discovered field: in this case, the wrapper, which initially equals page 1, is generalized by replacing string `'John Smith'` by `#PCDATA`. The same happens a few steps after for `'Database Primer'` and `'XML at Work'`.

It is worth noting that constant strings in the two pages, like `'Books of:'` at token 2, do not originate fields in the wrapper. These are rather considered as additional information added by the generating script as part of the HTML layout.

Tag Mismatches: Discovering Optionals Tag mismatches are used to discover iterators and optionals. In the presence of such mismatches, our strategy consists in looking for repeated patterns (i.e., patterns under an iterator) as a first step, and then, if this attempt fails, in trying to identify an optional pattern. Let us first discuss how to look for optionals based on tag mismatches (iterators are discussed in the next section). Consider Figure 3. The first tag mismatch occurs at token 6, due to the presence of an image in the sample and not in the wrapper. This image should therefore be considered as optional. To solve the mismatch, suppose first a search for a possible iterator has been done using the techniques that will be described in the next paragraph, and that this search has failed. We may therefore assume that the tag mismatch is due to the presence of optionals. This means that, either on the wrapper or on the sample we have a piece of HTML code that is not present on the other side, and that, by skipping this piece of code, we should be able to resume the parsing. This is done in two main steps:

1. Optional Pattern Location by Cross–Search With respect to the running example, given the mismatching tags at token 6 – `<UL>` and `<IMG.../>` – we know that: (*a*) assuming the optional pattern is located on the wrapper, after skipping it we should be able to proceed by matching the image on the sample with some successive `<IMG.../>` tag on the wrapper; (*b*) on the contrary, assuming the pattern is located on the sample, we should proceed by matching token 6 on the wrapper with the first occurrence of tag `<UL>` on the sample. A simple cross-search of the mismatching tags allows to conclude that the optional pattern is located on the sample (the wrapper does not contain any images from which to resume the parsing).

2. Wrapper Generalization Once the optional pattern has been identified, we may generalize the wrapper accordingly and then resume the parsing. In this case, the wrapper is generalized by introducing one pattern of the form `( <IMG src=.../> )?`, and the parsing is

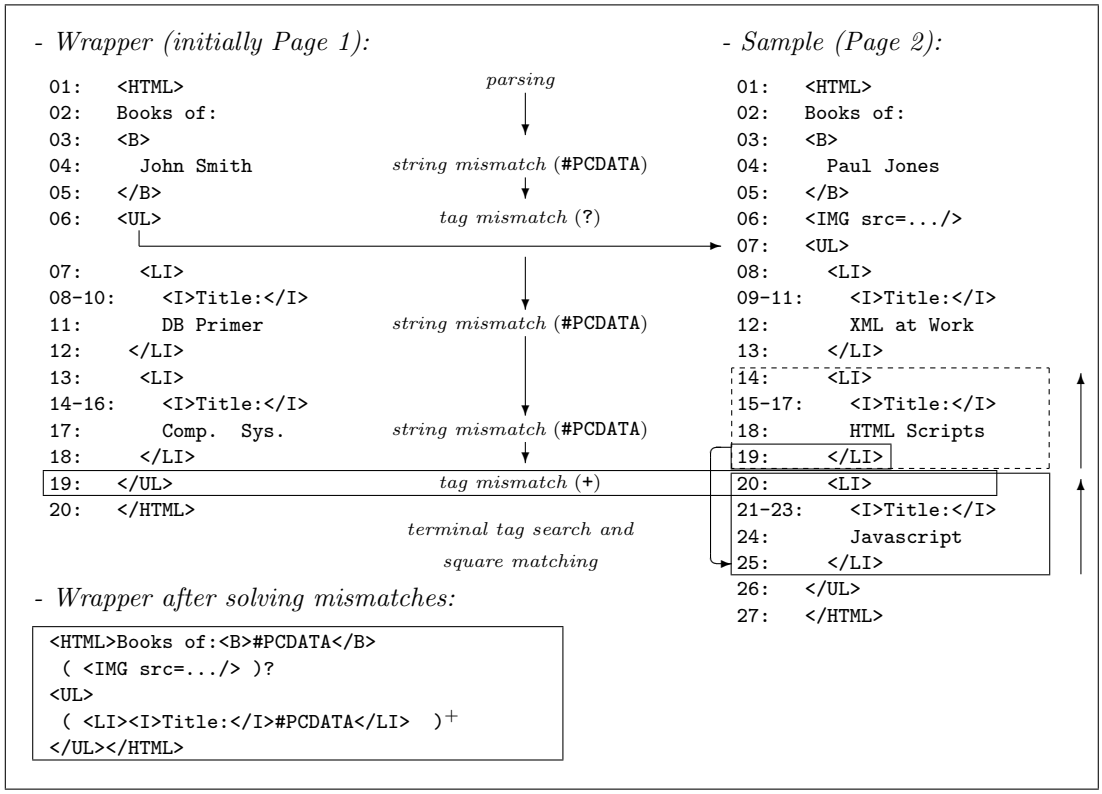


Figure 3: One Simple Matching

resumed by comparing tokens 6 and 7 respectively.

Tag Mismatches: Discovering Iterators Let us now concentrate on the task of discovering iterators. Consider again Figure 3; it can be seen that the two HTML sources contain, for each author, one list of book titles. During the parsing, a tag mismatch between tokens 19 and 20 is encountered; it is easy to see that the mismatch comes from different cardinalities in the book lists (two books on the wrapper, three books on the sample), i.e., of the repeated pattern `<LI><I>Title:</I>#PCDATA</LI>`. To solve the mismatch, we need to identify these repeated patterns that we call *squares*, and generalize the wrapper accordingly; then, the parsing can be resumed. The mismatch solving algorithm in this case goes through three main steps:

1. Square Location by Terminal–Tag Search After a tag mismatch, a key hint we have about the square is that, since we are under an iterator (+), both the wrapper and the sample contain at least one occurrence of the square. Let us call o_w and o_s the number of occurrences of the square in the wrapper and in the sample, respectively (2 and 3 in our example). If we assume that occurrences match each other, we may conclude that before encountering the mismatch the first $min(o_w, o_s)$ square occurrences have been matched (2 in our example).

As a consequence, we can identify the last token of the square by looking at the token immediately before the mismatch position. This last token is called *terminal tag* (in the running example, this corresponds to tag `</LI>`). Also, since the mismatch corresponds to the end of the list on one sample and the beginning of a new occurrence of the square on the other one, we also have a clue about how the square starts, i.e, about its *initial tag*; however, we don't know exactly where the list with the higher cardinality is located, i.e., if in the wrapper or in the sample; this means that we don't know which one of the mismatching tokens corresponds to the initial tag (`</UL>` or `<LI>`). We therefore need to explore two possibilities: (*i*) candidate square of the form `</UL>...</LI>` on the wrapper, which is not a real square; or (*ii*) candidate square of the form `<LI>...</LI>` on the sample. We check both possibilities by searching first the wrapper and then the sample for occurrences of the terminal tag `</LI>`; in our example, the search fails on the wrapper; it succeeds on the sample. We may therefore infer that the sample contains one candidate occurrence of the square at tokens 20 to 25.

2. Square Matching To check whether this candidate occurrence really identifies a square, we try to match the candidate square occurrence (tokens 20–25) against some upward portion of the sample. This is done backwards, i.e., it starts by matching tokens 25 and 19, then moves to 24 and 18 and so on. The search

succeeds if we manage to find a match for the whole square, as it happens in Figure 3.

3. Wrapper Generalization It is now possible to generalize the wrapper; if we denote the newly found square by s, we do that by searching the wrapper for contiguous repeated occurrences of s around the mismatch region, and by replacing them by $(s)^+$, as it is shown in Figure 3.

Once the mismatch has been solved, the parsing can be resumed. In the running example, after solving this last mismatch the parsing is completed. We can therefore conclude that the parsing has been successful and we have generated a common wrapper for the two input HTML pages.

Based on this example, it should be apparent why, in case of tag mismatches, we always look for iterators first. In fact, with respect to the mismatch between tokens 19 and 20, if we had first looked for an optional, we could have successfully produced a common wrapper in which each page contained two books, with a third optional book, and would have missed the list.

4.2 More Complex Examples

In Figure 3, the algorithm succeeds after solving several string mismatches and two simple tag mismatches. In general, the number of mismatches to solve may be very high, and each may represent a more challenging case than the ones discussed so far. In this section we discuss some of these challenges. Since string mismatches are relatively straightforward to handle, we will concentrate exclusively on tag mismatches.

– *Recursion:* Note, to start, that the mismatch solving algorithm is inherently recursive, since, when trying to solve one mismatch, more mismatches can be generated and have to be solved. To see this, consider Figure 4. Here, we have reported a further variant of the pages about authors: the pages have a nested structure, with a list of books, and for each book a list of editions. We start matching the sample (page 2) against the wrapper, which initially equals page 1. The parsing stops at token 15, where a tag mismatch is found. When trying to solve the mismatch looking for a possible iterator, we do the following: (i) based on the possible terminal tag (`</LI>` at token 14), we first locate one candidate square occurrence on the wrapper (tokens 15–28); then (ii) we try to match this candidate square against the upward portion of the wrapper. Remember that we match the square backwards, i.e., we start by comparing the two occurrences of the terminal tag (tokens 14 and 28), then move to tokens 13 and 27 and so on.

This comparison has been emphasized in Figure 4 by duplicating the wrapper portions that have to be matched. Since they are matched backwards, tokens are listed in reverse order. Differently from the previous example – in which the square had been matched by a simple alignment – it can be seen that, in this case, new mismatches are generated when trying to match the two fragments. These mismatches are called *internal mismatches*. The first internal mismatch in our example involves tokens 23 and 9: it depends on the nested structure of the page, and will lead us to discover the list of editions inside the list of books.

We may deal with these internal mismatches exactly in the same way as we do with external mismatches. This means that the matching algorithm needs to be recursive, since, when trying to solve some external mismatch, new internal mismatches may be raised, and each of these requires to start a new matching procedure, based on the same ideas we have discussed above, the only difference being that these recursive matchings don't work by comparing one wrapper and one sample, but rather two different portions of the same object.

With respect to the case in Figure 4, the external mismatch will trigger two internal mismatches. The first one, as discussed above, will lead to discover the list of book editions; the second one will lead to identify the optional pattern `<I>Special!</I>`. The final wrapper is also reported in the Figure.

– *Backtracking:* Another source of complexity in the algorithm comes from the need to choose between several alternatives which are not guaranteed to lead to a correct solution. When, going ahead in the parsing, these choices prove to be wrong, it is necessary to backtrack them and resume the parsing from the next alternative. To see why this happens, consider for example the discovery of iterators after a tag mismatch (the same ideas also hold for optionals). The first step to solve the mismatch requires to locate, on either the wrapper or the sample, one candidate square occurrence, and then try to match that internally. In this process, we are forced to choose among several alternatives: first we need to search for squares both on the wrapper and on the sample. Second, when we try to locate the square by a search of the terminal tag, we need to consider different occurrences of the terminal tag, which identify different candidate squares.

4.3 Matching as an AND-OR Tree

Based on the ideas discussed above, we can now give a more precise description of algorithm *match*. The algorithm works on one wrapper, w, and one sample, s, and tries to generalize the wrapper by matching it with the sample; this is done by parsing the sample using the wrapper, and by trying to solve all mismatches that are encountered during the parsing.

Finding one solution to $match(w, s)$ corresponds to finding one visit for the AND-OR tree [19] shown in Figure 5. In fact: (i) the solution to $match(w, s)$ corresponds to the solution of all external mismatches encountered during the parsing (AND node); solving each of these mismatches corresponds in turn to

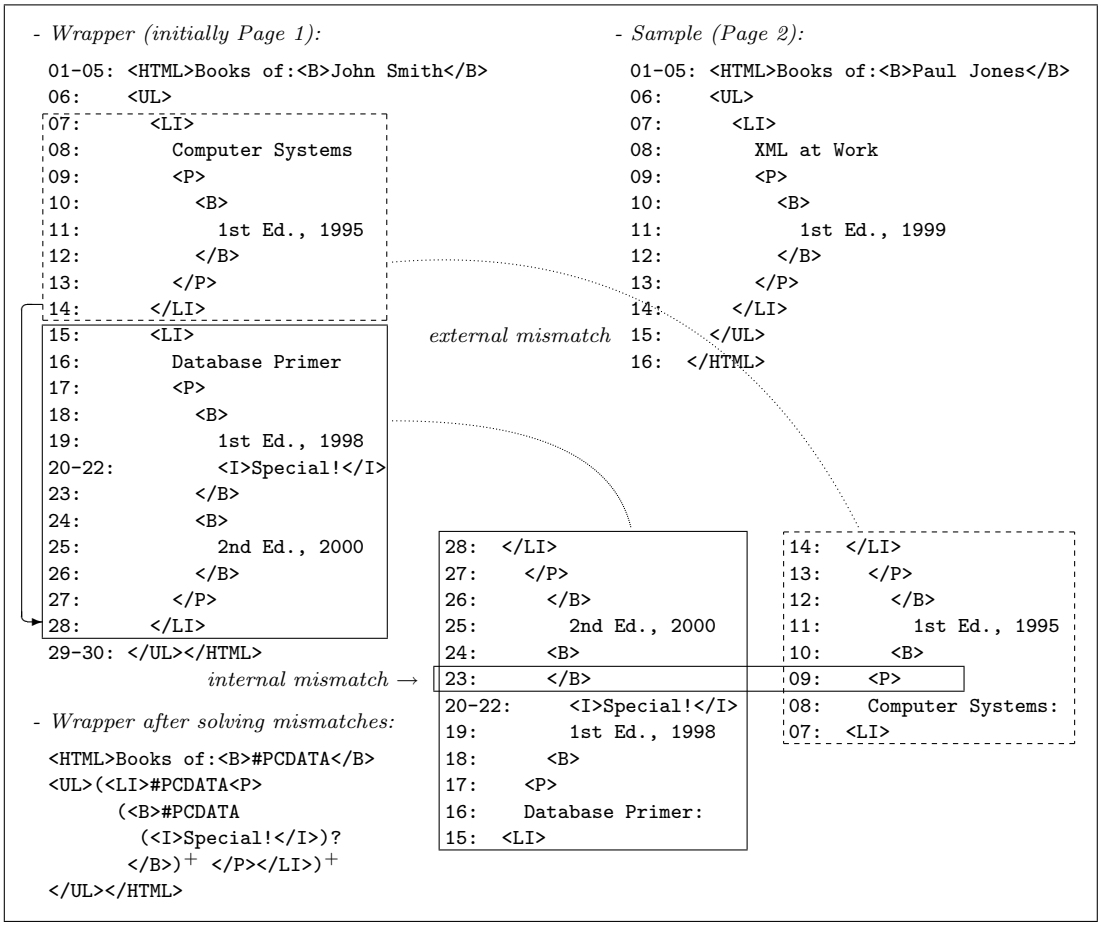


Figure 4: A More Complex Matching

finding one visit of an AND-OR subtree; in fact, (*ii*) the mismatch may be solved either by introducing one field, or one iterator, or one optional (OR node); (*iii*) the search may be done either on the wrapper or on the sample (OR); (*iv*) for both iterators and optionals there are various alternative candidates to evaluate (OR); (*v*) in order to discover iterators it may be necessary to recursively solve several internal mismatches (AND), each of which corresponding to a new AND-OR subtree. Given a wrapper w and one sample s, if we manage to solve the AND-OR tree, we return the generalized wrapper $w' = match(w, s)$ as output; otherwise the output is ϵ.

4.4 Lowering the Complexity

A formal argument (which we omit here for space reasons) shows that algorithm $match(w, s)$ has exponential time complexity with respect to the input lengths; intuitively, this depends on the fact that the AND-OR tree of a matching has in the worst case exponential size due to the need to explore different alternatives for each mismatch.

A primary goal of our implementation has been that of limiting the complexity of the match; to do this, we had to introduce several pruning techniques, in order to skip subtrees that most likely don't correspond to meaningful solutions. Based on these techniques, we were able to achieve a good compromise between expressibility and complexity, as shown by the running times reported in our experimental results (see Figure 6 in Section 5).

1. Bounds on the fan–out of OR nodes We put a constant bound k on the fan-out of OR nodes, i.e., on the maximum number of candidate square and optional occurrences to evaluate. This is largely justified by our experiences with real sites, which tell that both in cases of squares and optional, the right candidate is always very close to the mismatch point (in our implementation $k = 4$, although the system very seldom needs to evaluate more than one candidate occurrence). We therefore sort the children of each OR node based on the length of the candidate patterns, keep only the shortest k and discard the others.

2. Limited backtracking for external mismatches To reduce the amount of memory used to store the tree, we discard some portions of the tree for which a visit

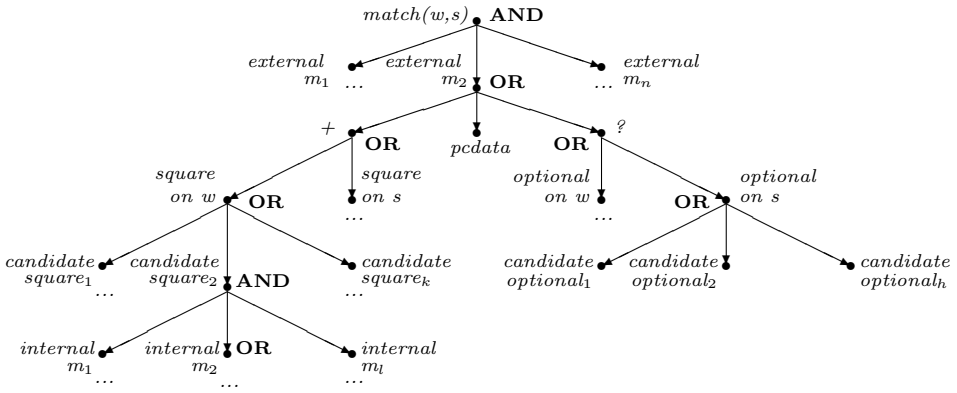


Figure 5: AND-OR Tree of the Wrapper Generation Problem

has been found. These subtrees correspond to external mismatches that lead to discover an iterator: in essence, since it is very unlikely that external mismatches lead to find out false iterators, we consider each of these discoveries as a fixpoint in the matching that will not be backtracked.

3. Delimiters Finally, in order to find a reasonable compromise between expressiveness and complexity of the matching process, we have decided to impose some constraints on the position of optional patterns in our wrappers; as a consequence, we further prune the tree based on the respective position of iterators and optionals in the wrapper: we discard all visiting strategies corresponding to wrappers in which a pattern (iterator or optional) is delimited on either side by an optional pattern (like, for example in `((<HR>)?<LI>#PCDATA</LI>)`$^+$, where the left delimiter of the pattern under `+` is `(<HR>)?`, or like in `(<BR>)?(<HR>)?`).

Although these choices slightly reduce the expressiveness of the formalism, they have the advantage of preventing, in practical cases, the generation of exponential searches, as it is confirmed by the low computing times and memory requirements exhibited by the system in our experiments, as discussed in the following section.

5 Experimental Results

Based on algorithm *match(w,s)* described above, we have developed a prototype of the wrapper generation system and used it to run a number of experiments on real HTML sites. The system has been completely written in Java. In order to clean HTML sources, fix errors and make the code compliant with XHTML, and also to build DOM trees, it uses *JTidy*, a Java port of *HTML Tidy* (`http://www.w3.org/People/Raggett/-tidy/`), a library for HTML cleaning. The prototype has been used to conduct experiments on several sites. All experiments have been conducted on a machine equipped with an Intel Pentium III processor working at 450MHz, with 128 MBytes of RAM, running Linux (kernel 2.2) and Sun Java Development Kit 1.3.

When it is run on a collection of HTML pages, it tries to iteratively apply algorithm *match* to generate a common wrapper for the pages. The algorithm is initialized by taking any of the pages as an initial wrapper. Then, at each successive step, it tries to match the wrapper generated at step before with a new sample. Since we are not in general guaranteed that the pages can all be described using a single wrapper, the algorithm may produce more than one wrapper. For each wrapper, we also want to store the set of *matching samples*, i.e., the list of pages from which the wrapper was generated. In this way, the algorithm will generate a collection of wrappers, and cluster the input HTML pages with respect to the matching wrapper.

We report in Figure 6 a list of results relative to several well known data-intensive Web sites. In each site, we have selected a number of classes of fairly regular pages; for each class we have downloaded a number of samples (usually between 10 and 20). Figure 6 actually contains two tables; while Table A refers to experiments we have conducted independently, in Table B we compare for 5 page classes our results with those of other data extraction systems for which experimental results are available in the literature, namely Wien [13, 12] and Stalker [14], two wrapper generation systems based on a machine learning approach.

Table A in Figure 6 contains the following elements: (*i*) *class:* a short description of each class, and the number of samples considered for that class; (*ii*) *results:* results obtained from the matching, i.e., number of wrappers (*#w*) created by the system, number of samples matching each wrapper (*#s*), outcome of the data extraction process (*extr*) – i.e., whether it was possible to actually extract a dataset from the pages or not – and overall computing time needed to examine the samples (the total time needed to compute matchings between samples in a given class; it does not include all times related to preprocessing the sources – i.e., calls to JTidy and tokenization, neither the time

Table A

classes			results				schema				
n.	site	description	#s	#w	#s	extr.	time	nest	pcd	opt	lists
1	amazon.com	cars by brand	21	1	21	yes	0"266ms	1	8	0	1
2	amazon.com	music bestsellers by style	20	-	20	no	-	-	-	-	-
3	buy.com	product subcategories	20	1	20	yes	1"107ms	2	16	0	4
4	buy.com	product information	10	1	10	yes	0"735ms	1	14	3	2
5	rpmfind.net	packages by name	30	1	10	yes	4"827ms	3	5	2	3
6	rpmfind.net	packages by distribution	20	1	20	yes	1"963ms	2	8	1	3
7	rpmfind.net	single package	18	2	10	yes	0"299ms	1	26	3	5
					8	yes	0"167ms	1	25	3	3
8	rpmfind.net	package directory	20	-	20	no	-	-	-	-	-
9	uefa.com	clubs by country	20	1	20	yes	0"434ms	1	5	2	1
10	uefa.com	players in the national team	20	1	20	yes	0"260ms	2	2	1	2

Table B

site			schema				comparative results					
n.	name (URL)	#s	pcd	nest	opt	ord	RoadRunner		Wien		Stalker	
11	Okra (discontinued)	20	4	1	no	no	√	0'0"700ms	√	5'2"	√	19'4"
12	BigBook (bigbook.com)	20	6	1	no	no	√	0'0"781ms	√	1h23'50"	√	7'4"
13	La Weekly (laweekly.com)	28	5	1	yes	no	√	0'0"391ms	no		√	1h08'
14	Address Finder (iaf.net)	10	6	1	yes	yes	no		no		√	3h22'1"
15	Pharmweb (pharmweb.net)	10	3	2	yes	no	√	0'0"350ms	no		no	

Figure 6: Experimental Results

needed for data extraction); (*iii*) *schema:* some elements about the structure of the dataset, namely: level of nesting (*nest*), number of attributes (*pcd*), number of optionals (*opt*) and number of lists.

As it can be seen from the table, for a large majority of pages the system was able to generate a wrapper and use that to extract a dataset from the HTML sources. This process was completely automatic and required no human intervention. Computing times are generally in the order of a few seconds; our experience also shows that the matching usually converges after examining a small number of samples (i.e., after the first few matchings – usually less than 5 – the wrapper remains unchanged). As it can be seen from the table, in some cases the system was unable to extract any data from the pages. There are two main sources for these behaviors, namely: (*i*) limited expressive power of union–free regular grammars; (*ii*) restrictions imposed in our implementation on the matching algorithm. These are discussed in the following.

Expressive Power In some of the cases UFREs are not sufficiently expressive to wrap the pages. This may happen either because the samples in a class form a non-regular language, or because they form a regular language which requires the use of unions to be described.

The first case – non-regular languages – is quite infrequent. The only case we have encountered in our experiments are package directories on `rpmfind.net`: these pages describe the folders in which a software package is organized; the nesting of folders is represented on the screen by progressive indentations of the folder names; as a consequence, tags in the corresponding pieces of HTML code form a language of nested balanced parenthesis, which is a well known context-free language and cannot be described using a regular grammar; therefore, our system fails in finding a common wrapper for the pages.

The second case – regular languages that require unions – is more frequent. One example of this kind are music bestsellers on `amazon.com`: these pages contain a list of items, some of which have customer reviews, some others don't; the HTML code in the two cases is different. As a consequence, our system is unable to discover repeated patterns in the list, and the wrapper generator fails (being unable to factorize the list, the system returns a number of wrappers in the order of the number of samples examined).

Implementation Choices Some other classes have been partitioned due to our choices in implementing the system. As discussed above, we disallow adjacencies between iterators and optionals. In some cases, these would be needed in order to generate a single wrapper for the pages. This happens, for example, for pages about single packages on `rpmfind.net`: the description of these pages requires a UFRE with an iterator adjacent to an optional pattern.

5.1 Comparison with other works

To compare our results with those of Wien and Stalker, Table B in Figure 6 reports a number of elements with respect to 5 page classes for which experimental results were known in the literature [12, 14]; the original test samples for classes 11 to 15 have been downloaded from RISE (`http://www.isi.-edu/~muslea/RISE`), a repository of information sources from data extraction projects. Table B contains the following elements: (*i*) *site* from which the pages were taken, and number of samples; (*ii*) description of the target *schema*, i.e., number of attributes (*pcd*), level of nesting (*nest*), whether the pages contain optional

elements (*opt*), and whether attributes may occur in different orders (*ord*); (*iii*) *results:* results obtained by the three systems, with computing times; times for Wien and Stalker refer to CPU times used during the learning.[1]

A few things are worth noting here with respect to the expressive power of the various systems. (*i*) While Wien and Stalker generated their wrappers by examining a number of labeled examples, and therefore the systems had a precise knowledge of the target schema, RoadRunner did not have any a priori knowledge about the organization of the pages. (*ii*) Although computing times refer to different machines, it can still be seen that, in those cases in which all three systems are able to generate a wrapper (11 and 12), CPU times used by RoadRunner are orders of magnitude lower than those needed to learn the wrapper both by Wien and Stalker. (*iii*) Differently from RoadRunner and Stalker, Wien is unable to handle optional fields, and therefore fails on samples 13, 14 and 15. (*iv*) Stalker has more considerable expressive power since it can handle disjunctive patterns; this allows for treating attributes that appear in various orders, like in Address Finder (14); being limited to union–free patterns, RoadRunner fails in cases like this. (*v*) Both Wien and Stalker cannot handle nested structures, and therefore they fail on PharmaWeb, the only class whose pages contain a list of lists (nest equals 2); on the contrary, RoadRunner correctly discovers the nesting and generates the wrapper.

5.2 Quality of the Extracted Datasets

An important comment is related to the quality of the data extracted by the wrappers. Although the dataset usually corresponds to the one expected by inspecting the pages, it is in some cases influenced by the physical HTML layout imposed to data in the pages.

To give one simple example, consider the pages listing clubs by country on `uefa.com`. From the logical viewpoint, each of these pages contains a list of club names, each with the relative city. However, for presentation purposes, this list is presented as a table with four columns, so that each row in the table contains data about two different clubs; in essence, we may say that the HTML layout induces a "physical schema" that can be interpreted as a list of quadruples *(clubName, city, clubName, city)*. When the wrapper generator runs on these pages, it has no clue about the fact that the four columns of the HTML table might be collapsed to two, and therefore generates a dataset of the form `(#PCDATA, #PCDATA, #PCDATA, #PCDATA)`$^+$.

In essence, in these cases all relevant data are correctly extracted, but according to a schema that is not the one expected after looking at the logical organization of the pages.

[1]Since for some of the pages Wien and Stalker consider only a portion of the HTML code, to have comparable results when needed we have restricted our analysis to those portions only.

References

[1] B. Adelberg. NoDoSE – a tool for semi-automatically extracting structured and semistructured data from text documents. In *SIGMOD'98*.

[2] P. Atzeni and G. Mecca. Cut and Paste. In *PODS'97*.

[3] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *VLDB'97*.

[4] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 23(8), 1998.

[5] D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, Y. Ng, D. Quass, and R. D. Smith A conceptual-modeling approach to extracting data from the web. In *ER'98*.

[6] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5), 1967.

[7] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3), 1978.

[8] S. Grumbach and G. Mecca. In search of the lost schema. In *ICDT'99*.

[9] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proc. of the Workshop on the Management of Semistructured Data*, 1997.

[10] C. Hsu and M. Dung. Generating finite-state transducers for semistructured data extraction from the web. *Information Systems*, 23(8), 1998.

[11] G. Huck, P. Frankhauser, K. Aberer, and E. J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *CoopIS'98*.

[12] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118, 2000.

[13] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *IJCAI'97*.

[14] I. Muslea, S. Minton, and C. A. Knoblock. A hierarchical approach to wrapper induction. In *Proc. of Autonomous Agents*, 1999.

[15] L. Pitt. Inductive inference, DFAs and computational complexity. In K. P. Jantke, editor, *Analogical and Inductive Inference, Lecture Notes in AI 397*. Springer-Verlag, Berlin, 1989.

[16] B. A. Ribeiro-Neto, A. Laender, and A. Soares da Silva. Extracting semistructured data through examples. In *CIKM'99*.

[17] A. Sahuguet and F. Azavant. Web ecology: Recycling HTML pages as XML documents using W4F. In *WebDB'99*.

[18] S. Soderland. Learning information extraction rules for semistructured and free text. *Machine Learning*, 34(1–3), 1999.

[19] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1979.