Automatically Regenerating Wrappers for Web Sources Using Results from Previous Queries

Juan Raposo, Alberto Pan, Manuel Álvarez
Department of Information and Communications Technologies. University of A Coruña.
{jrs,apan,mad}@udc.es

Justo Hidalgo
Denodo Technologies Inc.
jhidalgo@denodo.com

ABSTRACT
A substantial subset of the web data follows some kind of underlying structure. Nevertheless, HTML does not contain any schema or semantic information about the data it represents. A program able to provide software applications with a structured view of those semi-structured web sources is usually called a wrapper. Wrappers are able to accept a query against the source and return a set of structured results, thus enabling applications to access web data in a similar manner to that of information from databases. A significant problem in this approach arises because web sources may experiment changes that invalidate the current wrappers. In this paper, we present novel heuristics and algorithms to address this problem. Our approach is based on collecting some query results during wrapper operation. Then, when the source changes, they are used to generate a set of labeled examples that are provided as input to a wrapper induction algorithm able to regenerate the wrapper. We have tested our methods in several real-world web data extraction domains, obtaining high accuracy in all the steps of the process.

INDEX TERMS: [H.2.5.a] Data Translation, [H.2.8.m] Web Mining.
1 INTRODUCTION

A substantial subset of the web data follows some kind of underlying structure. For instance, web search forms are usually a front-end to a database where data is structured. These web sources are usually called *semi-structured*.

In many cases, semi-structured web sources contain very valuable information that cannot be easily accessed by other means. For instance, data may be provided by an autonomous organization which only provides web access for its information. In other occasions, directly accessing the backend database may be difficult or inconvenient.

A program able to provide software applications with a structured view of a semi-structured web source is usually called a *wrapper*. Wrappers are able to accept a query against the source and return a set of structured results, thus enabling applications to access web data in a similar manner to that of information from databases. Therefore, wrappers are a key part of most web data integration applications. Web wrapper generation has been an active research field for some years. See [8] for a survey.

The main problem with wrappers is that they can become invalid when the web sources change their underlying templates. This is called the *maintenance problem*. In this paper, we present novel heuristics and algorithms for dealing with this issue.

Our approach is based on the *wrapper induction* techniques for wrapper generation. In these systems, wrappers are generated by providing the system with a set of user-labeled examples which are employed to learn the underlying structure of the target HTML pages. Our approach for automatic maintenance collects some query results during wrapper operation and, when the source changes, they are used as input to generate a new set of labeled examples that can then be used to generate a new wrapper (i.e., once they have been labeled, the collected results play the role of the user-provided examples during the initial wrapper generation). We have experimented with our techniques in several important real-world web data extraction problems and have found them to be highly accurate.
The rest of the paper is organized as follows. Section 2 provides its context by presenting the basic stages in which the problem of wrapper maintenance can be divided in our approach. Section 3 describes the methods used to select the query results that must be stored during wrapper operation. Section 4 and 5 are the core of the paper and contain our main contributions. Section 4 presents the novel heuristics and algorithms that constitute our approach for generating the new set of examples from the stored query results and section 5 overviews our wrapper induction algorithm for regenerating the wrapper from the new set of examples. Section 6 describes our experiments with several real-world web sources. Section 7 discusses related work and section 8 summarizes our conclusions.

2 THE STAGES OF WRAPPER MAINTENANCE IN OUR APPROACH

The approach we use for wrapper generation and maintenance is based on the wrapper induction techniques ([4], [7]) for wrapper generation. In these techniques, wrappers are generated from a set of user-labeled examples.

The basis of our approach for automatic maintenance is making wrappers collect some results from valid queries during their operation, and when the source changes, to use those results to generate new labeled examples to bootstrap the wrapper induction process again. Thus, the process is comprised of four stages:

1. Collecting results from queries. During wrapper operation, the system stores the results from some selected queries.

2. Verifying Wrappers. It consists of verifying whether the wrapper continues to be valid or not.

3. Generating a new set of labeled examples. When the wrapper is determined to be no longer valid due to a change in the source, the system must generate a new set of labeled examples from the results stored during the first stage.

4. Re-generating Wrappers. From the examples generated in the previous stage, the new wrapper must be induced. The same induction algorithm used for the first wrapper generation
may be used here. In fact, other wrapper induction algorithm could be used for this purpose provided that the examples are expressed in the particular notation used by the induction method.

Although our implemented system deals with all these stages, in this paper we will not focus on the verification stage (this problem has been dealt with in works such as [5],[9]).

3 COLLECTING RESULTS FROM QUERIES

The objective of this stage is to collect the results from a subset of the queries executed by the wrapper during its normal operation so, when the source changes, they can be used to generate a new training set of labeled examples. The system works as follows:

1. When the wrapper answers a query, it stores up to a certain (customizable) number of results from the query in a local database. For each result collected, the system stores the query that generated it, the extraction rule (or set of rules) used by the current wrapper to parse it and an expiration date computed from current time.

2. Periodically, the current content in the database is explored for checking whether the non-expired results are sufficient to satisfy certain requirements we will describe below or not. If so, all the expired results are removed. If not, then the system computes a minimum set of expired queries that must remain stored.

The requirements needed in order to assure that the stored results satisfy are:

1. Storing at least a certain (customizable) number of query results. In general terms, the stages 3 and 4 will provide better output results when using a sufficient number of results (but not big enough to cause too much delay). We usually store at least 100 results.

2. Storing results from at least a certain (customizable) number of different queries. Queries may implicitly determine some properties of the response pages. For instance, an electronic shop may assign a given discount (e.g. 15%) for all the books by a given author \(a\). If we happened to use only results from the query ‘author=a’, the wrapper
induction process could wrongly consider the string “15%” to be an invariant for all the books. We usually store results from at least 10 different queries.

3. Storing a minimum (customizable) number of results for each current visual representation of a result in the source. Most web sources use slightly different visual representations for the results of a query. This means that a wrapper will need a different set of extraction rules for extracting the results using each visual representation. In order to generate the extraction rules for all the visual representations, the wrapper induction process will benefit of having examples from all of them. We have heuristically found very useful to store several results using each former visual representation (before the source change) in order to increase the probability that the stored results also cover all the new visual representations (after the source change).

4 GENERATING A NEW SET OF LABELED EXAMPLES

In this section we describe our approach for obtaining new labeled examples from the stored results.

First, let us introduce the data model underlying our system. We model the data elements to extract from a web source as a set whose members are either atomic label/value pairs or other elements. Atomic values are character strings. For instance we can model the schema of data elements of type ALBUM as the set \{TITLE, ARTIST, DATE, EDITION: \{FORMAT, PRICE\}\}. This way, a data element of a given type can be seen as a tree whose leaves are atomic values and the non-leaf nodes represent non-atomic elements (see Fig. 1).

This section is organized as follows: section 4.1 states the labeling problem and explains some of its difficulties. Section 4.2 describes the basic heuristics and algorithms of our approach for the base case where the data to be extracted may be modeled as a type exclusively composed of atomic fields. Section 4.3 generalizes these techniques to deal with types of arbitrary depth.
4.1 The problem of correctly labeling the examples

In the labeling stage, we have the following inputs:

1. A source modeled according to a type $T=\{field_1, \ldots, field_r\}$.
2. The set of queries $Q=\{q_1, \ldots, q_n\}$ whose results were collected during wrapper operation.
3. For each $q_i \in Q$, we will use as input its stored results. We will consider them as a set of unlabeled examples (or examples for short) $E_i=\{e_{i1}, \ldots, e_{im}\}$
4. For each $q_i \in Q$, the page or set of pages $P_i$ that form the current response to $q_i$ in the source.

The objective of this stage, to be fully explained later, is to label the maximum possible number of examples in the new pages. Labeling an example $e_{ij}$ consists in locating in $P_i$ an occurrence of the example by correctly identifying the values of all its constituent data fields.

For instance, let us consider the case of an electronic bookshop whose results are of type $T=\{\text{title}, \text{author}, \text{format}, \text{price}\}$. Let us also suppose an example $e_{ij} = \{(\text{title}, \text{"Beginning XML, Second Edition"}), (\text{author}, \text{"David Hunter et al"}), (\text{format}, \text{"Paperback"}), (\text{price}, \text{"27.19"})\}$ obtained through a query $q_i$. 

Fig. 1. ‘Album’ type tree

Fig. 2. Amazon result page

Fig. 3. “Fake” example
Correctly identifying the occurrence of $e_{ij}$ in a page $P_i$ may be more difficult than it could seem at first sight. For instance, consider Fig. 2, where an Amazon page containing $e_{ij}$ is shown. As can be seen, the values of the data fields of $e_{ij}$ appear in several positions on the page, making it difficult to identify the real occurrence of the example. For instance:

- The value of the ‘Format’ field (‘Paperback’) is repeated several times in the page, since many other books have the same value (the same may also apply to the ‘Price’ and ‘Author’ fields).

- Amazon shows information about the book represented by the example in two different places in the page (the list of most popular results and the complete list of results). Nevertheless, we should use the occurrence in the complete list, since we want our examples to help in generating extraction rules to extract all the results and not only the most popular ones.

Another complication arises because the stored example might actually not appear in the page even though the page contains individual occurrences of all of its data values. Since there will be some time interval between the moment when the results were collected and the moment when we try to regenerate the wrapper (usually a few hours), the data on the source may have changed during the interval. For instance, Fig. 3 shows an example where the price of the book has changed but, by chance, another book has the same price. Therefore, our system needs to detect these situations and discard these “fake” examples in order to provide the re-induction stage only with valid examples.

Note that the main assumption of our approach is that we can usually be confident that a good number of the unlabeled examples contained in $E_i$ are still present in $P_i$. It should be noted that in the cases where the values of the data elements to be extracted from the source vary very rapidly (e.g. stock quotes information), this assumption may be partly wrong. In section 4.2.5 we will describe how our approach can be extended to deal with those sources.
4.2 Labeling examples of types composed of atomic fields

Once we have stated the labeling problem, we proceed to describe our approach to address it. In this section, we will focus in the particular case where the data elements to extract are modeled according to a type composed exclusively of atomic fields. Following section will handle the more general case which deals with types of arbitrary depth.

More formally, we can define the problem of correctly labeling an example in this case as follows. Let us assume we have an example \( e \) of type \( T = \{field_1, \ldots, field_r\} \) which we want to label in the page \( P \). Let us also assume that all the fields of \( T \) are atomic. Then examples of type \( T \) are of the form \( e = \{(field_1, value_1), \ldots, (field_r, value_r)\} \) where \( value_k \) \( k=1..r \) is a string denoting the value for the \( k\)-th field of \( T \) in \( e \). Each string \( value_k \) \( k=1..r \) may appear multiple times in \( P \) and we will term each appearance of \( value_k \) as a candidate field occurrence of \( field_k \) for \( e \).

Now, a candidate example occurrence for \( e \) in \( P \) will be of the form \( co_e = \{(value_1, pos_1), \ldots, (value_r, pos_r)\} \) where \( (value_k, pos_k) \) \( k=1..r \) denotes the \( pos_k\)-th occurrence in \( P \) of the string \( value_k \). \( co_e \) will be a correct labelling for \( e \) iff the identified occurrences for all the \( value_k \) \( k=1..r \) form a real apparition in \( P \) of the data element represented by \( e \).

As we have already said, we have as inputs a source modeled according to a type \( T \) along with the set of queries \( Q \) whose results were collected during wrapper operation. For each \( q \in Q \), we will have a set of unlabeled examples \( E = \{e_1, \ldots, e_n\} \) and the page or set of pages \( P \) that constitute the current answer to \( q \) in the source.

An important remark is that if \( P \) consists of several pages (e.g. when the examples are contained in “detail pages” accessed from a search results listing, or when the result listing is paginated), the algorithms which will follow will consider all the input pages as if they were part of a single “aggregated page”. The DOM tree of this “aggregated page” would be built by making the DOM trees of the original pages all be children of a new common root node (i.e. the DOM trees of the original pages will be siblings).
The algorithm to label an example \( e_i \) in \( E \) in \( P \) is shown in Fig. 4. It works by computing all the candidate example occurrences for \( e_i \). If their number is higher than a certain (customizable) limit, then the most promising ones are chosen. Then the algorithm ranks them according to two heuristics: the “sibling nodes” heuristic and the proximity heuristic. The sibling nodes heuristic ranks a candidate example occurrence according to whether it is an instance of a repetitive sequence containing several other examples with similar layout or not. The proximity heuristic ranks candidate example occurrences according to the proximity in the page of the data values that form the \( m \).

The algorithm uses the sibling nodes heuristic as the main criteria for choosing the best candidate occurrence. The proximity heuristic is used only when the previous heuristic returns the same value for several candidates. This is because, as we will see, the sibling nodes heuristic has into account the context provided from all the other examples, thus making it very effective.

**Algorithm:** Obtain best atomic candidate occurrences for an example.

\[
R_i = \text{obtainBestAtomicCandidateOccurrences} \left( P, \mathcal{T}, E, e_i, \text{maxCandidates} \right)
\]

**Input:**
- \( P \), a document or a set of documents considered as if they were part of a single page where their DOM trees were siblings.
- A type \( \mathcal{T} = \{\text{field}_1, \ldots, \text{field}_r\} \), whose fields are all atomic.
- \( E = \{e_1, \ldots, e_n\} \), where \( e_j \) is an example data element of type \( \mathcal{T} \) contained in \( P \).
- \( e_j \) is of the form \( \{(\text{field}_1, \text{value}_{j1}), \ldots, (\text{field}_r, \text{value}_{jr})\} \) where \( \text{value}_{jk} \) denote the value for the \( k \)-th field of \( \mathcal{T} \) in \( e_j \).
- \( e_i \) is an example contained in \( E \).
- \( \text{maxCandidates} \) is the maximum number of candidate example occurrences to be considered in the algorithm.

**Output:**
- \( R_i \), a set with the best candidate example occurrences for \( e_i \).

1) Compute the number of occurrences of each \( \text{value}_{k1} \) in \( P \). This will output a set \( \{O_{ik}\}_{k=1}^r \) where \( O_{ik} \) is the set of occurrences of \( \text{value}_{ik} \) in \( P \).
2) Compute the number of all candidate example occurrences of \( e_i \) in \( P \) as \( \text{numCandidateExampleOccurrences} = \sum_{k=1}^r |O_{ik}| \).
3) A candidate example occurrence for \( e_i \) is of the form \( e_{oi} = \{(\text{value}_{i1}, \text{pos}_{i1}), (\text{value}_{i2}, \text{pos}_{i2}), \ldots, (\text{value}_{ir}, \text{pos}_{ir})\} \) where \( (\text{value}_{ik}, \text{pos}_{ik}) \) denotes the \( \text{pos}_{ik} \)-th occurrence in the page of the string \( \text{value}_{ik} \) and, thus, implicitly identifies a node in the DOM tree of \( P \).
4) If \( \text{numCandidateExampleOccurrences} > \text{maxCandidates} \) then:
   a) Choose the best candidate example occurrences guaranteeing \( \text{numCandidateExampleOccurrences} \leq \text{maxCandidates} \) (see section 4.2.3).
5) Sort the candidate example occurrences by using the “sibling nodes heuristic”.
6) If the “sibling nodes heuristic” is applicable (it returns a value greater than zero for some candidate occurrence of some example) and all the candidate example occurrences have value zero for this heuristic, then discard the example.
7) If the best candidate occurrences of any two examples overlap in the page, then discard the example with lower ranking according to the proximity heuristic.
8) Sort the occurrences with the same value for the sibling nodes heuristic by using the “proximity heuristic”.
9) Set \( R_i \) a set with all the occurrences with the best punctuation.

**Fig. 4. Base algorithm**
The following sub-sections describe in detail several aspects of the algorithm: section 4.2.1 details the sibling nodes heuristic and section 4.2.2 describes the proximity heuristic. Section 4.2.3 addresses the problem of pre-selecting candidate occurrences when their number is too high to compute the heuristic values for all of them. The section 4.2.4 deals with the detection of “fake” examples. Finally, section 4.2.5 describes the extensions needed to our approach to deal with the case of sources having data fields which varies very rapidly in time.

4.2.1 Sibling Nodes Heuristic

This heuristic builds on the observation that the data elements to extract from a semi-structured website are usually instances of a repetitive sequence whose elements share a similar layout and are placed in “symmetrical” positions inside the same page or in different pages.

Therefore, if a given candidate example occurrence $e_{o_i}$ is a real occurrence of an example $e_j$, then in the DOM tree of P there should be occurrences of other examples $e_k \in E$ in the “sibling subtrees” of the sub-tree containing $e_{o_i}$ (see Fig. 5).

This heuristic also applies when each page contains only one data instance (e.g. the detail page for each book in an electronic bookshop) simply by considering a single “aggregated page” (built as mentioned in the previous section) containing all the data instances.

The sibling nodes heuristic take as input the DOM tree of a page P and a candidate occurrence $e_o$ for the example $e=\{(field_1, value_1), (field_2, value_2), \ldots, (field_r, value_r)\}$

$e_o$ is of the form $e_o=\{(value_1, pos_1), (value_2, pos_2), \ldots, (value_r, pos_r)\}$, where $(value_k, pos_k)$ denotes the $pos_k$-th occurrence in the page of the string $value_k$ and, thus, implicitly identifies the
node in the DOM tree of $P$ where such occurrence begins (note that the occurrence could expand through several nodes).

We define the nearest common ancestor of $eo$ as the deepest common ancestor of all the nodes identified by $(\text{value}_1, \text{pos}_1), (\text{value}_2, \text{pos}_2), \ldots, (\text{value}_r, \text{pos}_r)$.

The sibling nodes heuristic tells us that if a given candidate example occurrence $eo_i$ is a real occurrence of the example $e_j$, then there is a high probability that a “sibling node” (a node with the same Xpath route from the root) of the nearest common ancestor of $eo_i$, contains a candidate occurrence of some other example $e_k$ of the same type.

Now, given a page $P$, a set of examples $\{e_1, \ldots, e_n\}$ for the same type $T=\{\text{field}_1, \ldots, \text{field}_r\}$, and a set of candidate example occurrences $\{eo_1, \ldots, eo_m\}$ for a given $e_j \in \{1, \ldots, n\}$ we can rank the example occurrences for $e_j$ using the following process:

1. For each $eo_i \in \{eo_1, \ldots, eo_m\}$:
   a. $\text{numSiblingOccurrences}_i = 0$
   b. Obtain $A_i$, the nearest common ancestor for $eo_i$.
   c. Obtain $\{S_1, \ldots, S_t\}$ the set of sibling nodes of $A_i$ in the DOM tree of $P$ (including $A_i$).
   d. For each $S_j, j=1..t$:
      i. For each $e_k \in \{e_1, \ldots, e_n\}$, search for candidate occurrences of $e_k$ in the sub-tree with root $S_j$.
      ii. If candidate occurrences are found for exactly one $e_k$, then
          (1) If any of the candidate occurrences found has “similar layout” to that of $eo_i$ then
              $\text{numSiblingOccurrences}_i ++$.

2. Sort candidate example occurrences in $\{eo_1, \ldots, eo_m\}$ by $\text{numSiblingOccurrences}_i$ in descending order.

A crucial consideration about the algorithm is the method used in step 1.d.ii.(1) for searching candidate example occurrences with “similar layout” to $eo_i$. The basic idea is to retain only those
having their values positioned in the tree in a similar way as $eo_i$. So, we compute the Xpath route for every field occurrence of each candidate example occurrence, and we compare them with the route to the corresponding field occurrence in $eo_i$, checking if they are equal (certain tags such as $<B>$ or $<I>$ are considered irrelevant for these purposes). In addition, we check that the ordering of the field occurrences is the same as in $eo_i$.

It should also be noted that the step 1.d.ii of the algorithm increments $numSiblingOccurrences_i$ if and only if the found candidate example occurrences from a node belong to exactly one alternative example. If candidate occurrences are found for more than one example, then it probably means that we are searching from a node which is too “high” in the DOM tree. This usually identifies a situation like the one shown in Fig. 6, where our candidate example occurrence is actually composed of field occurrences from several different data elements. Thus, in these cases, the heuristic ranks the candidate occurrence with value 0.

4.2.1.1 Applicability of the sibling nodes heuristic.

The sibling-nodes heuristic cannot be applied in all cases. In some rare situations the DOM tree of a page is constructed in such a way that any common ancestor node to all the data fields of an example is, at the same time, ancestor of the data fields of the other examples (see Fig. 7). In these situations, the nearest common ancestor will always be considered as too high by the sibling nodes heuristic and it will always return zero for every occurrence of every example even
if some occurrences are actually valid. So, in these cases the sibling nodes heuristic is not applicable and the proximity heuristic will be used.

4.2.2 Proximity Heuristic

This heuristic builds on the observation that all the data values belonging to a certain valid example tend to be grouped near each other in the page. In conformance with this, the candidate example occurrences whose values are sparse in the page should be considered less promising.

We can measure the “sparseness” of a candidate example occurrence, $co$, by computing the sum of the distances between every two consecutive field occurrences belonging to $co$. The distance between two field occurrences is computed as the number of nodes between the last node of the first occurrence and the first node of the next occurrence (depth-first traversal order).

Another possibility for implementing this heuristic is actually computing the average of the “visual distance” in the page rendered by the browser between the fields of the candidate example occurrence. The visual distance between two DOM nodes may be computed by using the functionalities provided by current browser’s APIs. In our tests, the DOM distance measure performs almost as well as the visual distance “measure”. Thus, for convenience and efficiency reasons, our current implementation uses the DOM distance measure.

4.2.3 Dealing with too many candidate occurrences

Sometimes the number of candidate occurrences of an example is too high and it is not possible to compute the heuristic values for all of them. In these cases the algorithm needs to make a first pre-selection among all the possible combinations of the candidate field occurrences of the example, to select in advance the most promising ones (step 4.a in Fig. 4). See Appendix A for a formal description of this algorithm.
4.2.4 Detecting “fake” examples

As it has already been said, it may occur that a chosen example is not really present in the page although there are candidate field occurrences for all of their values (recall Fig. 3). This is dealt with in step 6 of the obtainBestAtomicCandidateOccurrences algorithm, where examples that have a zero value in the “sibling nodes heuristic” for all their candidate example occurrences are rejected. This allows detecting the kind of ”fake” examples as the one shown in Fig. 3, since its nearest common ancestor will be “too high” in the DOM tree (recall section 4.2.1) for all their candidate occurrences.

Nevertheless, this rule must only be applied when the “sibling nodes” heuristic is applicable (recall section 4.2.1.1 for a description of the cases in which it is not) because, in other case, all the examples would be wrongly discarded. We consider the sibling nodes heuristic as not applicable if all the candidate occurrences rank as 0 in the sibling nodes heuristic for all the examples. In those cases, the proximity heuristic is used.

Another technique we apply for detecting “fake” examples is analyzing overlaps in the page among the best candidate occurrences of several examples. More precisely, if the best candidate occurrences for two examples overlap, we reject the example whose best candidate occurrence has a lower rank according to the proximity heuristic. The intuition here is that sources do not show the fields of a data element mixed with the fields of other data element. The criterion to choose the example to retain is again based on the observation that sources tend to show the fields of the same data element near to each other.

4.2.5 Dealing with Highly Time-Variable Data

When data which is to be extracted from a source vary at very short time intervals (e.g. stock quotes), we cannot assume any input example extracted from previous queries to remain completely valid in that source.
In these situations, the system distinguishes between the fields which are highly variable (e.g. the current change for a stock quote) and those which we can assume that will remain stable in at least some cases (e.g. the company acronym). For “highly variable” fields we will compute the candidate field occurrences by using a regular expression describing the format of the values for the fields (e.g. \[0-9\]+ \.” \[0-9\]+ for the current change of a stock quote). We also allow including in the expression which prefixes and suffixes are expected for the data values. As we will remark in the experience section of the paper, the techniques described in the previous sections remain effective.

4.3 Generalization to Types of Arbitrary Depth

In this section we show how to generalize the *obtainBestAtomicCandidateOccurrences* algorithm to examples of types of arbitrary depth.

Let us consider the tree for an example with depth \( k \) (see Fig. 8a). At the \((k-1)th\) level, all the non-atomic data instances of the example are composed exclusively by atomic values. Thus, they can be considered as if each of them were an example having type of depth 1. For instance, in the Fig. 8a the data elements (format= ”CD”, price= ”11.85”) from Example 1, and (format= ”LP”, price= ”17.34”) from Example 2, may be considered as the unlabeled examples *EX1.EDITION1* and *EX2.EDITION2*. We will call them as “subexamples of \((k-1)th\) level”.

Thus, we can consider every input example, obtain their sub-examples of \((k-1)th\) level and apply the *obtainBestAtomicCandidateOccurrences* algorithm with them.

It is important to note that the algorithm may return more than one candidate occurrence for each sub-example. For instance, in the Fig. 8a the candidate occurrences marked as \(co_1\) and \(co_2\) will be both returned as “best candidate occurrences” for the sub-examples *EX1.EDITION1* and *EX2.EDITION1*.

Now, for every example, we will substitute in \(P\) the minimum HTML code containing each “best candidate occurrence” of the sub-examples of \((k-1)th\) level by a special markup unique in the
Fig. 8a. General algorithm operation

Fig. 8b. General algorithm operation

page. If two candidate occurrences are formed by exactly the same data values then this markup will be equal for them, as shown in the Fig. 8b for \( co_1 \) and \( co_2 \).

Performing this process for the \((k-1)th\) level sub-examples of all the input examples, the best candidate occurrences for non-atomic sub-elements of level \( k-2 \) can then be computed by simply invoking \texttt{obtainBestAtomicCandidateOccurrences} with this representation of \( P \) and using the special unique markups as values of new atomic fields of \((k-2)th\) level, which substitute the non-atomic fields.
For instance, in the Fig. 8b (EX1.EDITION[1]="$EDITION1", EX1.EDITION[2]="$EDITION2", EX1.EDITION[3]="$EDITION3") and (EX2.EDITION[1]="$EDITION4", EX2.EDITION[2]="$EDITION4") will be the atomic fields which substitute the non-atomic field EDITION in the example EX1 and EX2 respectively.

It is important to note how the ambiguity we had at (k-1)th level (where the candidate occurrences co1 and co2 were both considered as “best candidate occurrences” for both EX1.EDITION1 and EX2.EDITION1) will be solved at this level: the context provided by the other edition fields and the other fields of the example allows the sibling-node and proximity heuristics to determine co1 as the only “best candidate occurrence” for EX1 and co2 as the only “best candidate occurrence” for EX2.

Repeating this process until the 0th level, we will obtain the best candidate occurrences for the complete examples. The algorithm is sketched in Fig. 9.

Algorithm: Obtain best candidate occurrences for a set of examples E.

– R = obtainBestCandidateOccurrences (P,T,E,maxCandidates)

Input:

– P, a document or a set of documents considered as if they were part of a single page where their DOM trees were siblings.
– E={e1, …en}, where ei is an example data element of type T contained in P.
– maxCandidates is the maximum number of candidate example occurrences to be considered in the algorithm at each level.

Output:

– bco = the set containing the best candidate occurrences for the examples e1,…en.

13) Build R the tree of T.
14) k=depth(R).
15) While (k>0)
   a) k=k-1
   b) bco = Ø (the empty set)
   c) For every sub-example of k-th level se of every example ej?
      i) SE={all subexamples of the same type of se in any example contained in E}
      ii) Obtain bco_se = obtainBestAtomicCandidateOccurrence (P,SE, se, maxCandidates), the set of the best candidate occurrences for se.
      iii) bco = bco U bco_se
   d) if (k>0)
      i) Substitute in P each eo ? bco by an unique markup generated in function of the data values that form it. If two candidate occurrences eo1 and eo2 are formed by the same data values, then the markup generated for them will be equal.
      ii) Substitute in every ej ? E every non-atomic field of level k by as many atomic fields as sub-examples has the non-atomic field, and assign as value to each added atomic field, the markup generated for the corresponding sub-example.
16) Return bco.

Fig. 9. General algorithm
5 WRAPPER INDUCTION

In this section we overview our algorithm to regenerate the wrapper. In the section 5.1, we provide a quick introduction to DEXTL (which stands for Data Extraction Language), the wrapping language we use (see [12] for detail). In the section 5.2 we describe the induction process which takes care of how the DEXTL programs are automatically generated from a set of labeled examples.

5.1 DEXTL Overview

A DEXTL program is composed of hierarchically structured DEXTL elements. Typically, a program for extracting data elements of a given type \( T \) will have a DEXTL element for each field from \( T \). The DEXTL elements will also be hierarchically arranged in the same way that the fields they represent are related in \( T \).

Thus, each DEXTL element can be either atomic (those without sub-elements) or non-atomic (those with sub-elements). Each non-atomic element has (among others not relevant here) the following parts: EXTRACTION clause (mandatory) and UNTIL clause (optional).

The EXTRACTION clause relates an element with its sub-elements and specifies how they are laid in the target source pages. This clause will be the basis to identify and extract the element data instances in the document. The UNTIL clause delimitates the end of the region of the document where the occurrences of the element must be searched for.

The basic structure used to specify the EXTRACTION and UNTIL clauses is called a DEXTL pattern. As we will see, a DEXTL pattern defines a sequence of data elements and separators among them.

The EXTRACTION clause will typically contain one DEXTL pattern for each possible visual layout of the element inside the target pages. The UNTIL clause uses DEXTL patterns to locate the end of the search region for the element. When the system is extracting instances of an element and an occurrence is found for a DEXTL pattern from its UNTIL clause, the system
stops looking for occurrences of that element and continues searching for occurrences of its
parent element (or stop if there is not a parent element).

5.1.1 DEXTL Patterns

A DEXTL pattern is comprised of a list of text tokens which are laid consecutively in the
document and which are delimited by tag-separators.

Text tokens represent text in the browser-displayed page. They are enclosed between '[' and ']' and they can be divided into portions by applying Perl5-like regular expressions. A placeholder (prefixed by the '$' character) can be assigned to the parts of the regular expression enclosed between '(' and ')'. The name may correspond either with an atomic field of the elements we wish to extract or with a special value called IRRELEVANT, which is used to represent non-constant strings appearing in the pattern, but which we do not wish to extract.

Tag-separators represent a regular expression concerning HTML document tags. For instance, we could define a tag-separator called EOL (EndOfLine) as follows: EOL = ("<br>"| "</p>"| "</tr>" | "</td>" ([ 
\r\t])* </tr>"

Though they can be defined to suit, DEXTL includes built-in tag-separators which are enough for the vast majority of situations concerning wrapper generation. Nevertheless, as we will see later, in the case of wrapper maintenance, tag-separators sometimes need to be dynamically built according to the characteristics of the pages we wish to deal with.

Each DEXTL pattern has a set of associated tag-separators, which are indicated through the SEPARATORS construction. All HTML tags that do not conform to a tag-separator are just ignored by the DEXTL extraction programs.

Example: Fig. 10 shows two search results from an electronic music shop. We wish to extract items having type ALBUM:{TITLE, ARTIST, DATE, EDITION:{FORMAT, PRICE}}. Fig. 11 shows the respective fragment of HTML code, where tag attributes have been omitted. Fig. 12 shows a DEXTL program to extract the occurrences of the element ALBUM.
In the EDITION extraction pattern the only tag-separator used is \textit{EOL}, while in the ALBUM extraction pattern the \textit{TAB} ("\textless \texttt{td}\textgreater") and \textit{ANCHOR} ("\textless \texttt{a}\textgreater") tag-separators are used.
5.2 Inducing DEXTL Patterns

Our induction algorithm will generate a set of DEXTL patterns for each non-atomic field of the type of the data elements we wish to extract. Then, it will compose a DEXTL program nesting the generated patterns according to the type structure. It will also find any UNTIL pattern needed.

Our basic algorithm for induction is \( \text{obtainAtomicPattern} \), which is shown in Appendix B. This algorithm receives as inputs a type \( T \) which is assumed to be exclusively composed of atomic fields, a set of examples for data elements of type \( T \) and a page \( P \) containing those examples (or, as usual, a set of pages considered as a unique page where their DOM trees are siblings, although...
this time we will not remove font tags). The algorithm outputs a set of DEXTL patterns for extracting occurrences of the desired elements of type $T$ from the target pages.

The `obtainAtomicPattern` algorithm iterates over the examples to generate a candidate pattern based on each example. The process stops when the set of current patterns recognizes all the examples. Then, the set of results matched by each pattern is explored to remove redundant patterns (those which only match results also extracted by other patterns). The final step of the algorithm consists in lengthening the pattern to make it as less ambiguous as possible while still recognizing the same examples. This is made by adding tokens from $P$ at the end and the beginning of the candidate pattern.

The following sub-sections describe in detail certain aspects of the algorithm: how candidate patterns are generated (5.2.1), how new tag separators are induced when needed (5.2.2) and how the algorithm is generalized to accept types of arbitrary depth.

### 5.2.1 Generating candidate patterns

The process for generating the candidate pattern for an example $e$ is shown in the algorithm of Fig. 13. The process is illustrated with an example in Fig. 14. The basic idea is as follows: first, we obtain an HTML portion containing the example and generate a preliminary DEXTL candidate pattern using the current set of tag-separators (we begin by using the former set of tag-separators used by the wrapper), and using the special placeholder IRRELEVANT to represent all the visible texts present in the HTML portion (Fig. 14 step 1).

Then, we execute the pattern on $P$. The set of matches is probable to contain occurrences of other examples (the examples having the same “visual layout” than $e$). For instance in the Fig. 14, the preliminary pattern generated for the first example also matches with the third one.

Therefore, we identify these matches and use them to refine the pattern in the following way:

- We consider the text tokens that correspond with the field occurrences of the other input examples enclosed. We substitute the field occurrence by a placeholder with its name, and
we find the longest common prefixes and suffixes of the strings surrounding the field occurrence. The common prefixes and suffixes will be considered as “fixed” in the refined regular expression for defining the text token. (see Fig. 14 step 2 for an example)

- Although it is not shown in the algorithm of Fig. B for the sake of brevity, the process of looking for common fixed prefixes and suffixes is also applied to the text portions which do not contain any field occurrence (using IRRELEVANT as placeholder for the variable parts), but only if the number of other examples matched by the pattern exceeds a certain parameter \( k \) (currently set to 5). This is a compromise to avoid generating “fixed” text sections that are too specific.
5.2.2 Inducing New Tag-Separators

The process gets more complicated if the current set of tag-separators is no longer valid when the source changes. This situation is detected in the step 3 of the algorithm for generating the candidate patterns. The system can correct this situation by applying the following rules:

1. The DEXTL interpreter assumes that the tag-separators cannot appear inside of the strings which match with the text tokens of the pattern (in other case, they would not actually be separators). Therefore, if there are any occurrences of a tag-separator inside an occurrence of a field of an example, then that separator must be removed from the list of tag-separators. For instance, in the Fig. 15 (step 1) the tag separator EOL is not longer valid because it splits in two parts the commentary.

2. Any pair of text tokens (field names or IRRELEVANT) in a DEXTL pattern must be separated either by a text (specified as a prefix or suffix in the regular expression associated to the text token for the field) or by a tag-separator. If this does not happen for a preliminary
DEXTL pattern, the system will add a new tag-separator by analyzing the HTML tags between both fields in all examples matching with the pattern, and choosing a common string of tags from either the beginning or the end (ignoring attributes), and ensuring that the chosen separator does not violate the former rule (it cannot appear inside of a field occurrence). For instance in Fig. 15 (step 2), after removing EOL as a valid tag-separator, the title and commentary of the news stories are not separated by neither a text nor a current tag-separator. Therefore, we need to generate a new tag-separator. “</A>” is a suitable election because it is a string of tags (of length 1) which appears between the title and commentary of the news stories but does not violate rule 1.

3. In the same way, the complete examples also need to be separated either by a text or by a tag-separator. If this does not happen for a preliminary DEXTL pattern, the system will add a new tag-separator in a similar way as in the previous case.

5.2.3 Generalization to types of arbitrary depth

The algorithm for the generalization of \textit{obtainAtomicPattern} to types of arbitrary depth can be consulted in Appendix C. The basic idea is reminiscent of the one used to generalize the algorithms for labeling examples in section 4.3. We recursively build candidate patterns for all the non-atomic sub-elements of the type and assemble them hierarchically to make up a DEXTL program. An additional difficulty for the generalized algorithm is computing the \textit{UNTIL} clauses that the DEXTL interpreter needs in order to detect the end of the region where the occurrences for a certain sub-element are found. This is overcome by searching a DEXTL pattern with zero matches inside the examples of the sub-element and which appears always after the last example of the sub-element.

6 EXPERIMENTS

To evaluate the effectiveness of our approach we monitored a set of Web sites from April 2004 to October 2004. We selected sources from different domains presenting different characteristics
and data types. For instance we included certain sources with some highly variable data fields (e.g. Nasdaq) for testing our techniques in that scenario. We also included some sources where the contents change frequently (e.g. New York Times).

For every web site we first generated a wrapper and used it on a daily basis to execute different queries. The system automatically stored some of the queries and their results according to the techniques explained in section 3. When a change in the format of the pages of a web site was detected (the changes were detected by manually checking the sources every day), the system generated a new set of labeled examples and induced the new wrapper. Once the wrapper was regenerated, we tested it with a new set of pages from the source obtained through different queries.

We will quantify the results at two different points of the regeneration process: after generating the new set of labeled examples and after the re-induction stage. At the first point we can measure the effectiveness of our techniques for generating new labeled examples. At the second point the wrapper is completely regenerated and we can measure the fitness of the generated examples to re-induce the wrapper using our induction algorithm.

To quantify the results of the stage of generating a new set of labeled examples we define the following metrics:

- LN: number of input examples (from the stored queries).
- LPE: number of input examples that are still present in the current HTML pages of the source after the change.
- LTE: number of total examples labeled by the system.
- LCE: number of examples correctly labeled by the system.
- Labeling Recall (LR) = LCE/LPE.
- Labeling Precision (LP) = LCE/LTE.

To quantify the results once the wrappers have been regenerated we define the following metrics:
N: number of data items which should be extracted from the test pages.

TE: number of total data items extracted by the regenerated wrapper from the test pages.

CE: number of correct data items extracted by the regenerated wrapper from the test pages.

Recall (R) = CE / N.

Precision (P) = CE / TE

6.1 Analysis of Results

Table 1 lists the subset of the monitored sources where any changes occurred during the period of study. Each of the sources underwent just one format change except for Barnes & Noble, Amazon Magazine, Espacenet and Yahoo People where two changes occurred (in these cases we computed the averages of the obtained results). The last column indicates if the sibling nodes heuristic was applicable after the change.

Some of the changes were relatively small. For instance, in Yahoo People a new column was added to the table that renders the people information. On the other hand, some sources completely changed their underlying templates (for instance, Barnes & Noble).

Table 2 shows the values of the metrics computed for the labeling process. A first conclusion is that the ratio of preserved examples is high for almost all the monitored sources. This supports
the idea of using results from previous queries as a good basis onto which generate a new set of examples for re-induction.

The only source where the ratio of preserved examples is very low is *New York Times*, in which only 19\% of the examples is preserved. In the sources with data fields considered as highly variable (*Nasdaq*, *Yahoo Quotes*, *AllBooks4Less*), we computed the percentage without considering those fields. As it can be seen, the ratio of preserved examples is very high, indicating that the values of the remaining fields in these sources are very stable.

The LR and LP metrics give a 100\% value for all the sources except for *AllBooks4Less* where the system failed to extract an example that was present in the page and, in addition, it did not discard two “fake” examples; and for *New York Times* where one example was wrongly labeled.

The errors in *AllBooks4Less* were caused because the sibling nodes heuristic was not applicable and there were some rare cases where the proximity heuristic failed. In *New York Times* the sibling nodes heuristic is applicable but only the 19\% of the stored results were present.

The errors in precision are caused by situations such as the one shown in the Fig. 16. In the figure, the system erroneously labels an occurrence of a book (which is not present in the source anymore) by using data fields really belonging to two other different books.

---

**TABLE 2**

**GENERATING NEW EXAMPLES METRICS.**

<table>
<thead>
<tr>
<th>Source</th>
<th>LN</th>
<th>LPE (%)</th>
<th>LR (%)</th>
<th>LP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>100</td>
<td>62 (1)</td>
<td>98.4</td>
<td>95.3</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>200</td>
<td>73.5</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>100</td>
<td>50</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>100</td>
<td>65</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>100</td>
<td>99</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td>87</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>200</td>
<td>98</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Nasdaq (Flash Quote)</td>
<td>100</td>
<td>100 (1)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>110</td>
<td>21</td>
<td>95.2</td>
<td>95.2</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100 (1)</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

(1) Without considering highly variable data fields
It is important to note that if the sibling nodes heuristic were applicable in this source, this situation would have been detected in the way discussed in the section 4.2.4. Even when the sibling nodes heuristic is not applicable, the situation would have been detected and corrected if the second of the two books involved in the Fig. 16 were present in our input examples (obviously, were the first book present, the proximity heuristic would have ranked the correct occurrence better than the “fake” one). Then, the proximity-based test for overlapping examples which was shown in section 4.2.4 would have discarded the false occurrence.

In New York Times the sibling nodes heuristic is applicable but only the 19% of the stored results were present. That increased the probabilities of failure.

It is important to note that the accuracy was only affected (an even then, only slightly) when the sibling nodes heuristic is not applicable or when the sibling nodes heuristic is applicable and the number of preserved examples is extremely low. For instance, there is another source in the set (Espacenet) where the sibling nodes heuristic is not applicable. Nevertheless, the results are still good because the percentage of preserved examples is high. On the other hand, the Amazon Magazine source preserves only 50% of the query results but the sibling nodes heuristic is applicable and, therefore, the quality of the labeling process is not affected. This is coherent with the expected behavior of the heuristics: the sibling-nodes heuristic is less vulnerable than the proximity heuristic to the lack of preserved examples.

Table 3 shows some metrics calculated after the wrappers were regenerated completely. The second and third columns show the values computed for the Recall and Precision metrics. The fourth column indicates the number of examples used to test the wrapper. The fifth column
shows the number of input examples to the re-induction algorithm (i.e. the examples labeled in the previous stage). In all the sources all examples are correctly labeled except in *AllBooks4Less* and *New York Times* where 3 of the 64 and 1 of the 21 examples are erroneous respectively. The sixth column shows the number of extraction patterns created by our induction algorithm for each wrapper. As we have already seen in section 5, an extraction pattern can be roughly matched to a different “visual representation” of the data items in the source.

The basic factor influencing the results in this stage is that the induction process benefits from having at least a certain number of examples for each possible visual representation, so the appropriate extraction rules can be induced.

So, the recall of the regenerated wrapper depends on the suitability of the set of labeled examples to cover the whole range of those different visual representations. If the examples do not cover this entire range then, even when the 100% of the examples are correctly labeled in the previous stage, the system will still not reach the 100% for the recall and the precision metrics.

Consequently with this, the worst results were obtained in *Amazon Book*, which is, by far, the source with more variety of visual representations, and in *New York Times*, which is the source with more data variability (only 19% of the stored examples are preserved).

**TABLE 3**

<table>
<thead>
<tr>
<th>Source</th>
<th>R (%)</th>
<th>P (%)</th>
<th>N</th>
<th>LTE</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllBooks4Less (Book)</td>
<td>99.7</td>
<td>97.9</td>
<td>1000</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Amazon (Book)</td>
<td>86.8</td>
<td>93.4</td>
<td>1000</td>
<td>147</td>
<td>27</td>
</tr>
<tr>
<td>Amazon (Magazine)</td>
<td>99.8</td>
<td>100</td>
<td>1000</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>Barnes&amp;Noble (Book)</td>
<td>98.2</td>
<td>100</td>
<td>1000</td>
<td>65</td>
<td>8</td>
</tr>
<tr>
<td>BigBook (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Cordis (Project)</td>
<td>97.8</td>
<td>100</td>
<td>1000</td>
<td>99</td>
<td>5</td>
</tr>
<tr>
<td>Delphion (Patent)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>Discoweb (Album)</td>
<td>100</td>
<td></td>
<td>1000</td>
<td>87</td>
<td>2/1 (1)</td>
</tr>
<tr>
<td>Espacenet (Patent)</td>
<td>99.5</td>
<td>100</td>
<td>1000</td>
<td>196</td>
<td>12</td>
</tr>
<tr>
<td>Nasdaq (Flash Quotes)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>NewYorkTimes (News Story)</td>
<td>90</td>
<td>100</td>
<td>200</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>YahooPeople (Person)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>YahooQuotes (Quote)</td>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>

(1) Album type / Edition subtype
In AllBooks4Less the precision also falls a bit because, as we have already seen, some “fake” examples were introduced in the labeling stage.

The source having worse results in the precision metric is Amazon Book. The precision fails are imputable in this case to the induction process and not to the labeling stage. Analyzing the source with more detail, we realized that most of the wrong matches generated by our induction algorithm were located in the portions of the pages containing the data items that the system missed to match (remember that recall was 86.8%). Thus, improving recall through better example coverage would also result in improving precision.

6.2 Time Complexity and Algorithm Optimizations

In our experiments the time it took to regenerate a wrapper (labeling and re-induction stages) ranged from 7 seconds (NewYorkTimes) to 3 minutes (Amazon Books) and the average time was 52 seconds.

In the worst case, the time complexity of both the labeling and re-induction stages is quadratic with respect to the number of input examples. The labeling stage is also influenced by the maximum number of candidate example occurrences considered for each example in the Base Algorithm (Fig. 4).

A first consideration is that, according to our experiments, both the number of examples and the number of candidate example occurrences considered are enough to provide high accuracy. So, we expect that the size of our input will not vary enough to cause this to be a problem in practice. In addition, our real implementation applies several optimizations that decrease the real complexity in the vast majority of cases.

- In the labeling stage, we cache the results of the search for examples performed in the step 1.d.i of the sibling nodes heuristic (see section 4.2.1). This way, if the search from a given node was already performed when labeling a previous example, it does not need to be repeated for other examples. Since many examples will usually have the same deepest
common ancestor (and therefore the same “sibling nodes”) this improvement is very significant in practice.

- As can be seen in section 5.2, the base algorithm for induction (shown in Appendix B) will usually not need to iterate over every example. The pattern generated for an example $e$ will also recognize other examples (the ones having the same “visual representation” as $e$). Since the process will stop when all input examples have been recognized by at least one pattern, it only would be needed to iterate over every example in the extremely rare case where each example needed a separate extraction pattern. So the actual dominating factor in execution times is the number of different “visual representations” of the source.

7 RELATED WORK

Wrapper generation has been an active research field for years (see [8] for a survey). These works are only concerned with wrapper generation and do not support automatic wrapper maintenance.

[5,9] address the wrapper verification problem and thus, are complimentary to our work.

[9,10,11] have addressed the problem of wrapper maintenance. [9] uses information collected during previous wrapper operation to perform an automatic process for labeling a set of new examples, thus producing the input for their induction algorithm. The process for identifying the correct candidate field occurrences does not have into account neither the context provided by the candidate occurrences of the other fields conforming the element, nor by the other examples of the same type. In addition, they do not deal with the maintenance of wrappers for pages containing “lists” of data elements. In our experience, the need for extracting information from this kind of pages is very common.

[10] identifies candidate field occurrences by assuming some structural features to remain invariant when sources change. These features are annotations (text strings that identify a certain data field, such as “Title” or “Price”), hyperlink features and syntactic features (it assumes that
the values will conform to a given regular expression). Then, they use the identified candidate field occurrences and the schema of the target data elements, to identify a set of blocks in the page which conforms to the structure and content suggested by the data schema. In sources where the assumption of invariance of annotations and hyperlinks does not hold the system will fail to identify correct examples.

[11] proposes a wrapper re-induction approach which iteratively uses the extracted data at a given time \( t \) to regenerate the wrapper at a later moment \( t+s \), where \( s \) is small. Their approach can only induce LR-wrappers [6] which are not expressive enough to deal with most modern web sites. In addition, they do not deal with the problem of correctly labelling the examples in the target pages after a source change.

Another difference with respect to [9,10,11] is that our system has into account the context provided by other examples in order to choose the best candidate occurrences for an example. This improves the accuracy of the labeling process and makes the approach less vulnerable when the target pages have several portions that conform to the underlying schema.

[13] proposes an ontology-based schema for wrapper generation. In this schema, an expert creates an ontology describing a particular domain of interest and the wrappers for sources compliant with the domain-description can be automatically generated by the system. In this approach the wrappers are resilient to changes as long as the ontology continues describing the source after the change. Defining such ontologies for non-trivial domains is a very challenging task since the differences between sources in the same domain can be vast.

The works presented in [1] and [3] deal with automatic wrapper generation and are also related with our system to some extent. They take as input a set of example pages of the same class, and automatically induce the underlying template they conform to. These approaches have the advantage of not requiring input examples for wrapper generation (at the cost of having less information on which to base the building of accurate wrappers), but they do not annotate the extracted results. While human post-annotation of the data is acceptable in wrapper generation, it
is not for automatic wrapper maintenance. In [2] it is presented an approach for automatic annotation of the extracted data using the texts surrounding the extracted items in the page. Nevertheless, it can not deal with the common situations where some data fields have not associated labels in the page.

8 CONCLUSIONS

In this paper we have presented techniques for the generation and automatic maintenance of wrappers for semi-structured web sources. Our approach is based on using inductive learning algorithms for generating wrappers. For automatic maintenance, our system collects some query results during wrapper operation and, when the source changes, they are used to generate a new set of labeled examples which constitute the input to induce the new wrapper.

The main step involved in this process is correctly labeling in the new pages the examples collected during previous wrapper operation. Our approach is based in two heuristics (“sibling nodes” and proximity) which arise from observations on the typical ways to dispose semi-structured information in a page or set of pages. We have experimentally tested our techniques for a variety of real-world web sources obtaining a high degree of effectiveness.

9 REFERENCES


APPENDIX A. DEALING WITH TOO MANY CANDIDATE OCCURRENCES

Algorithm: Obtain more promising atomic candidate occurrences for an example i.
– \( R_i = \text{obtainMorePromisingCandidateOccurrences}\left(\{O_{i1}, \ldots, O_{ir}\}, \text{maxCandidates}\right) \)

Input:
– The set of occurrences of each value \( v_k \) in \( P \) (value \( v_k \) represents the value of the \( k \)-th field in the i-th example). We will denote the set of occurrences of value \( v_k \) as \( O_k \).
– maxCandidates is the maximum number of candidate example occurrences to return.

Output:
– A set, \( R_i \), containing a maximum of ‘maxCandidates’ best candidate occurrences of example i.

1. Set \( R_i = \emptyset \)
2. Set field \( m \) the field with fewer occurrences. \( O_{im} \) will be the set with fewer occurrences: \(|O_{im}| = |O_{ik}|_{k=1..r} \)
3. For each occurrence of field \( m \), that will be of the form \((v_{im}, pos_j)\) \( j=1..s \)
   a. Set \( O_{imj} = \{(v_{im}, pos_j)\} \)
   b. For each field \( k \) \( k=1..r, k \neq m \) calculate \( O_{ikj} \) getting from \( O_k \) the occurrences with the deepest common ancestor with \((v_{im}, pos_j)\) from the root of the DOM tree.
   c. Set \( \text{minDepth} \) the minimum depth of all the common ancestors considered in the previous step.
   d. For each field \( k \) \( k=1..r, k \neq m \) add to \( O_{ikj} \) the occurrences of \( O_k \) with a deepest common ancestor with \((v_{im}, pos_j)\) with depth = \( \text{minDepth} \).
4. Calculate numCandidateExampleOccurrences = \( \sum_{j=1..s} \sum_{k=1..r} |O_{ikj}| \)
5. If numCandidateExampleOccurrences > maxCandidates
   a. Set particularMaxCandidates = maxCandidates / s.
   b. For \( j=1..s \)
      i. Calculate a reduction factor \( f \) such as \( \sum_{k=1..r, k \neq m} \text{Floor}(|O_{ikj}| * f) = \text{particularMaxCandidates} \)
         Note that if \( \text{Floor}(|O_{ikj}| * f) = 0 \) for some \( k \) and \( j \) then it will be considered as 1.
      ii. For \( k=1..r, k \neq m \)
         (1) Set numOccurrences = \( \text{Floor}(|O_{ikj}| * f) \)
         (2) Get from \( O_{ikj} \) the numOccurrences occurrences with smaller DOM distance from \((v_{im}, pos_j)\). Delete the rest of occurrences from \( O_{ikj} \).
6. Build the set of the best candidate occurrences for the example i, combining the occurrences of the fields selected in the previous steps. For \( j=1..s \)
   a. Obtain all the combinations of field occurrences of the sets \( O_{ik} \), to create candidate occurrences of the form: \((v_{i1}, pos_1) \ldots (v_{im}, pos_j) \ldots (v_{ir}, pos_r)\), where \((v_{im}, pos_j) \neq O_{ikj}\) \( k=1..r, k \neq m \)
   b. Add to \( R_i \), all the candidate occurrences generated in the previous step.

Obtain more promising atomic candidate example occurrences algorithm
APPENDIX B. BASIC ALGORITHM FOR INDUCTION

Algorithm: Generate a set of DEXTL patterns to extract occurrences of T.

\[\text{EP} = \text{obtainAtomicPattern}(P, T, S, E)\]

Inputs:
- P, a document or a set of documents considered as if they were part of a single page where their DOM trees were siblings.
- A type T exclusively composed of atomic fields, T={field_1, …, field_r}.
- S, the current set of tag type separators.
- E={e_1, …, e_n}, where e_i is a labeled example data element contained in P for the data element D. e_i is of the form \{(field_1, value_{i1}, pos_{i1}), …, (field_r, value_{ir}, pos_{ir})\} where (value_{ij}, pos_{ij}) denotes the pos_{ij}-th occurrence in P of the string value_{ij}, which is the value for the j-th field of T in e_i.

Outputs:
- EP={ep_1, …, ep_k} the set of generated extraction patterns.

1. EP=∅; UnmatchedExamples=E;
2. While (UnmatchedExamples ? ∅)
   a. Choose any e_i ? UnmatchedExamples.
   b. Generate a candidate pattern cp_i from e_i (see figure 12).
   \[\text{cp}_i = \text{generateCandidatePattern}(P, T, S, E, e_i)\]
   c. Compute the number m_i of matchings for cp_i in P, and the number me_i of matchings which correspond with some e_k ? E.
   d. Let MatchedExamples be the set of examples matched by cp_i. UnmatchedExamples = UnmatchedExamples – MatchedExamples.
3. End While
4. Now EP contains a set of extraction patterns ep_1, …, ep_k. Remove from EP the patterns such as all the examples recognized by them are recognized by other patterns.
5. Disambiguate the patterns of EP. For each pattern add tokens at the end and the beginning while it recognizes the same set of samples (me_i). Get as result the smaller patterns with the fewer (m_i – me_i).
6. EP contains the extraction patterns for T.

Basic algorithm for induction
APPENDIX C. GENERALIZATION OF THE BASIC ALGORITHM FOR INDUCTION TO TYPES OF ARBITRARY DEPTH

Algorithm: Generate a DEXTL program to extract occurrences of T.
- \( DP = \text{generateDEXTLProgram}(P, T, E) \)

Input:
- \( P \), a document or a set of documents considered as if they were part of a single page where their DOM trees were siblings.
- A type \( T = \{\text{field}_1, \ldots, \text{field}_r, \text{subtype}_{1,1}, \ldots, \text{subtype}_{1,s} \} \), where \( \text{subtype}_{1,j} \) is defined recursively as \( \{\text{field}_{1,1}, \ldots, \text{field}_{1,m(j)}, \text{subtype}_{j,1}, \ldots, \text{subtype}_{j,s(j)} \} \) \( j = 1, \ldots, s \)
- \( E = \{e_1, \ldots, e_n\} \), where \( e_i \) is a labeled example data element of type \( T \) contained in \( P \).

Output:
- \( DP \), a DEXTL program to extract occurrences of \( T \) in \( P \).

1. Build \( R \) the tree of \( T \).
2. If depth(\( R \)) \( > 1 \)
   - a. For every non-atomic sub-element of \( T \), \( \text{subtype}_{1,j} \) \( j = 1, \ldots, s \)
     i. Set \( E_j = \{e_{ijk} \mid k = 1, \ldots, t(ij)\} \) \( i = 1, \ldots, n \), \( k = 1, \ldots, t(ij) \)
   - b. \( DP_j = \text{generateDEXTLProgram}(P, \text{subtype}_{1,j}, E_j) \).
3. Generate independent sets of extraction patterns for each subset of atomic fields of \( T \) which are extracted between \( \text{subtype}_{1,j} \). If we have \( s \) \( \text{subtype}_{1,j} \), we will construct at most \( s + 1 \) subsets of fields of \( T \). We call this subsets \( T^m_{m=1,\ldots,s+1} \) \( \text{T}^1 \) and \( \text{T}^{s+1} \) could be empty.
   - a. Set \( S = \{\text{type tag separators associated to } T\} \).
   - b. For each \( T^m_{m=1,\ldots,s+1} \) \( \text{T}^1 \) and \( \text{T}^{s+1} \) could be empty.
     i. Set \( E_m = \{e_i \mid i = 1, \ldots, n \} \) \( e_i \) denotes the set of all the nested examples of \( \text{subtype}_{1,j} \) for all examples in \( E \) \( t(ij) \) is the number of sub-examples contained in \( e_i \) for \( \text{subtype}_{1,j} \)
     ii. \( EP_m = \text{obtainAtomicPattern}(P, T_m, S, E_m) \). \( EP_m \) will be the set of generated extraction patterns for “datatype” \( T_m \) and samples \( E_m \) in \( P \).
4. Calculate the patterns of the Until clause for each non-atomic subelement (UP).
   - a. For each \( T^m_{m=1,\ldots,s+1} \) \( \text{T}^1 \) and \( \text{T}^{s+1} \) could be empty.
     i. Set \( E = \{e_i \mid i = 1, \ldots, n \} \) \( e_i \) denotes the set of all the nested examples of \( \text{subtype}_{1,j} \) for all examples in \( E \)
     ii. Set \( EI \) = \( \emptyset \), \( UI \) = \( \emptyset \), \( L \) = \( \emptyset \)
     iii. For each \( e_i \) \( E \)
       (1) Add to \( EI \) the fragment of HTML code comprising from the previous example or subexample to the last \( e_{ijk} \mid k = 1, \ldots, t(ij) \)
       (i.e. the fragments where occurrences of \( \text{subtype}_{1,j} \) are searched).
       (2) Add to \( UI \) the fragment of HTML code after the last \( e_{ijk} \mid k = 1, \ldots, t(ij) \) and the next \( e_{ijb} \) (the next occurrence of \( \text{subtype}_{1,j} \) for any example \( e_i \) or the end of the document (i.e. the fragments where the until pattern should be matched).
       (3) Add to \( L \) the beginning position of the next example or subexample (i.e. the limit position where the until pattern can be matched).
     iv. \( UP = \text{searchUntilPattern}(EI, UI, L, T) \)
     v. Add \( UP \) as the UNTIL clause of the root element of \( DP \), the DEXTL program to extract from \( P \) items of \( \text{subtype}_{1,j} \).
5. Set \( DP \) an empty extraction program.
   - a. Add to \( DP \), in the correct order, the extraction patterns for each subset of fields of \( T \) \( EP_m \) and the root element of \( \text{subtype}_{1,j} \).
6. Return \( DP \).
Algorithm: Search the until pattern for a subtype \(j\).

- \(UP = \text{searchUntilPattern}(EI, UI, L, S)\)

Input:
- \(EI = \{ei_k\}_{k=1..s}\) is the set of fragments of HTML code where the occurrences of subtype \(j\) are searched.
- \(UI = \{ui_k\}_{k=1..s}\) is the set of fragments of HTML code where the until pattern of subtype \(j\) should be matched.
- \(L = \{l_k\}_{k=1..s}\) where \(l_k\) is the limit position where the until pattern can begin in the fragment \(ui_k\).
- \(S\), the set of tag type separators.

Output:
- \(UP\), the DEXTL until pattern for the subtype element \(j\).

1) Divide \(ui_1\) in tokens using the tag separators of \(S\), obtaining a list \(V = (\text{Token}_1, \ldots, \text{Token}_z)\) where \(\text{Token}_i\) is either a text token or a tag-separator token \(? S\).
2) Set \(UP = \emptyset\)
3) Set \(p\) a pointer initially pointing to \(\text{Token}_1\).
4) Set \(p’\) a pointer initially pointing to \(\text{Token}_1\).
5) While the beginning position of the token pointed by \(p\) < \(l_1\):
   a) Add the token pointed by \(p’\) to \(UP\)
      i) If it is a text token add the special placeholder \(\text{IRRELEVANT}\)
      ii) If it is a tag-separator token add the tag-separator
   b) If \(UP\) is not matched in \(ui_k\) for any \(k \in \{1..s\}\)
      i) Set \(UP = \emptyset\)
      ii) Advance \(p\) to the next token
      iii) \(p’=p\)
   c) Else
      i) Set \(MU\) the set of matches of \(UP\) in \(UI\)
      ii) If the beginning position of the match \(m_k\) begins after \(l_k\) for any \(k \in \{1..s\}\)
         (1) Set \(UP = \emptyset\)
         (2) Advance \(p\) to the next token
         (3) \(p’=p\)
      iii) Else
         (1) If \(UP\) is not matched in \(ei_k\) for \(k \in \{1..s\}\)
            (a) Return \(UP\).
            (2) Else
               (a) Construct \(UP’\) as a refined version of the pattern \(UP\), by searching common prefixes and suffixes for the tokens of type \(\text{IRRELEVANT}\) in \(UP\), using for that purpose the matches of \(MU\) (the process is identical to that explained in the steps 5 y 6 of the \text{generateCandidatePattern} algorithm).
               (b) If \(UP’\) is not matched in \(ei_k\) for \(k \in \{1..s\}\)
                  (i) return \(UP’\)
               (c) else
                  (i) Advance \(p’\) to the next token

Find UNTIL clause algorithm