Data Driven XPath Generation

Robin De Mol, Antoon Bronselaer, Joachim Nielandt, and Guy De Tré

Dept. of Telecommunications and Information Processing,
Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
Email: {robin.demol,antoon.bronselaer, joachim.nielandt, guy.detre}@UGent.be

Abstract. The XPath query language offers a standard for information extraction from HTML documents. Therefore, the DOM tree representation is typically used, which models the hierarchical structure of the document. One of the key aspects of HTML is the separation of data and the structure that is used to represent it. A consequence thereof is that data extraction algorithms usually fail to identify data if the structure of a document is changed. In this paper, it is investigated how a set of tabular oriented XPath queries can be adapted in such a way it deals with modifications in the DOM tree of an HTML document. The basic idea is hereby that if data has already been extracted in the past, it could be used to reconstruct XPath queries that retrieve the data from a different DOM tree. Experimental results show the accuracy of our method.

Keywords: XPath Generation, Data Driven, HTML

1 Introduction

With big data a hot topic at the moment, data extraction has been gaining a lot of attention over the last few years. In that field semi-structured data sources, such as HTML, present a large challenge due to their property of separating data from the structure that presents it. Semi-structured documents preserve a notion of hierarchy and dependencies between values but there is no tabular schema behind the structure. It is the hierarchy itself that supposedly provides metadata on the data.

Current data extraction tools, often referred to as wrappers, usually heavily rely on XPaths to identify where data of interest can be found. However, if the structure of a document is changed, the XPaths to the data change with it, which creates the need for a reconfiguration of the data extraction tool. With our approach we aim to improve existing tools by providing the functionality to automatically reconstruct XPaths to previously identified data.

First, we will extract data using an XPath based data extractor. Second, we will use this data to attempt to reconstruct the XPaths used to configure the data extractor.

The remainder of the paper is structured as follows. In Section 2, we explain why our approach is beneficial for extraction tools. In Section 3, we indicate how our methods can be placed in current technology. Section 4 provides some preliminaries for our work. In Section 5, we elaborate on our technique, illustrated

by an example that will be used throughout the rest of the work. In Section 6, we discuss how we can use data to construct XPaths. In sections 7 and 8, we explain how we can use these XPaths to configure a new wrapper. In section 9 we present a use case to measure the accuracy of our research. In Section 10, we indicate what might still improve our work and finally, in Section 11 we present our final conclusions.

2 Problem statement

XPath based data extraction tools show some shortcomings. One of their most important issues lies in the semi-structured nature of HTML. Because data and structure are separated, a purely structural approach to retrieve data carries inherent risk, because there is no way to verify if the given XPaths extract the correct data. When websites are generated through templates, generally it is true that similar data will be found in similar nodes, resulting in similar XPaths. However, because of a new visual update to the website, the template that generates the structure can be changed while the data remains the same, as to present it in a different fashion. Then, it can happen that the configured XPaths no longer point to data, or worse, point to the wrong data.

Therefore, a more intelligent approach is needed for data extraction from semi-structured documents, based on both structure and data. Under the assumption we can detect changes in the structure of an HTML document, we propose a technique to automatically reconstruct XPaths to the data of interest.

3 Related work

The internet has been profiled as the largest source of data available. Therefore, there has been a lot of research effort into the creation of data extraction tools [1, 2]. These are classified based on the degree of automation.

The most commonly represented tools are semi-automated, relying on a manual training phase after which the crawler can continue to run automatically [3–6]. Alternatively, there are fully automated techniques [7–9], which require a single URL as input and aim to return a set of objects extracted from the content. Though these tools are considered to be easier to use, their output generally requires more attention than that of semi-automated solutions. For the latter, the manual training phase often allows semantic annotation that is used to post-process the data, which is lost when using the former.

Due to the semi-structured nature of HTML, existing systems additionally almost exclusively use a structural approach. Most of these rely on the DOM tree representation, also called tag tree, of the document [4, 5, 7–11], though some also look at the way the data is visualized [3, 12] using optical recognition software. Such techniques often focus on frequency and pattern analysis of subtrees to estimate where data of interest can be found.

On the topic of wrapper breakage, which occurs when a wrapper can no longer retrieve data because the HTML document is changed structurally, there

has been less research [13–17]. These approaches discuss ways to identify data in labeled documents, but do not offer a solution for text analysis or entity recognition. Furthermore, these techniques often use a different than XPath based (re)configuration, reintroducing the need for another manual configuration.

Our approach can be classified as a semi-automated and part structural, part data driven technique for web data extraction. The novel factor of our methodology is the combination of both structural and data driven aspects, providing a strong, yet highly automated solution to reconfigure XPath based data extraction tools.

4 Preliminaries

4.1 XPath

XML Paths (XPaths) are used to uniquely identify one or more elements in XML documents. Due to the fact that HTML and XML are both based on Standard General Markup Language (SGML), they are highly similar. Therefore, XPaths can and have also be used to identify elements in HTML documents.

XPaths structurally heavily resemble resource locators from UNIX-based file systems and hence consist of a sequence of steps separated by forward slashes. XPaths can be absolute or relative. Absolute XPaths always starts with a forward slash and are evaluated starting from the root element, a required element in all SGML documents. Relative XPaths assume a context, usually an internal element, and describe a path starting from there.

XPaths can be used to navigate through HTML documents by executing them in sequence. A new navigation is always indicated by an absolute path. Subsequent XPath queries are relative to the results of the previous one. After each XPath query, the elements resulting from the navigation are returned and are used as a context for the next query.

In contrast to UNIX-based paths, XPaths offer additional functionality in the form of predicates, indicated by square brackets. Such predicates can be used to select specific elements from the results of an XPath query. Predicates can contain wildcards to point to multiple elements simultaneously.

To illustrate some of the possibilities, a small example is provided. Below is one of the numerous alternative ways to list the authors of this paper in HTML, followed by some simple XPaths that can be used to select data from it.

```
<html>
    <h1>Authors</h1>
    <p>Robin De Mol</p>
    <p>Antoon Bronselaer</p>
</html>

/html[1]/p[1] → selects author Robin De Mol
/html[1]/p    → selects all authors by omitting the index predicate
/html[1]/*    → selects all child elements of the html element
```

XPaths can be used in a broader context with more flexibility but this functionality lies outside the scope of this paper.

4.2 DOM

The Document Object Model (DOM) is a tree-based representation of an HTML document's structure consisting of nodes. Each node has at least a type and possibly a name or value.

The type, which can be `element` for elements and `text` for data, among others, indicates how the node should be interpreted. In case of an element node, the name represents the kind of element, such as `p`, `a` and `html`. In case of a text node, the name is "#text" [1].

The value contains the actual data in case of a text node and equals null in case of an element node. Nodes of the element type can contain any number of child nodes. Getting the text component of any node concatenates its own value with the text component of all its descendants.

Below is the DOM tree for the HTML document from the previous example.

```
{type: element, name: html, value: null}
    {type: element, name: h1, value: null}
        {type: text, name: null, value: Author list}
    {type: element, name: p, value: null}
        {type: text, name: null, value: Robin De Mol}
    {type: element, name: p, value: null}
        {type: text, name: null, value: Antoon Bronselaer}
```

5 General approach

It is assumed here that XPath queries are used to map HTML documents onto a relational database. Therefore, we have chosen to model our data in a tabular way. Hence, we assume that the data we want to extract consists of multiple records. Each record is a collection of one or more attributes. Among them we choose one attribute to be the key attribute. This attribute needs to be present for each record as it is used to identify it in the document and in the database. The other attributes are called relative attributes. If the HTML document is generated by a template, we assume that each relative attribute can always be found at the same relative location to the key data for each record. We use this assumption to model concepts like "X can be found below Y" or "A is located to the right of B".

Throughout the remainder of this work, we will use a case study of a bookstore: www.eci.be. We identify books as records. For each book, the title is the key attribute and the author is a relative attribute. Other candidates for relative attributes are the publisher, amount of pages, price and description, among others.

[1] `http://docs.oracle.com/javase/6/docs/api/org/w3c/dom/Node.html`

Fig. 1. Example data from http://www.eci.be/boeken/roman-literatuur/literatuur. Key attributes (titles) are marked in green, relative attributes (authors) in yellow.

First, we extract a data set of books from http://www.eci.be/boeken/roman-literatuur/literatuur using an XPath based data extraction tool. For this, we identify the attributes of interest, in our case title and author, and configure one XPath per attribute. For the key attribute, the title, we specify one absolute XPath which evaluates all book titles on the document. For the relative attribute, the author, we specify one relative XPath, which tells the wrapper where the author can be found, given the title. Our goal is to reconstruct these XPaths using our approach. First, we investigate how we can reconstruct the XPath for the key attribute, the title. Then, we look at how we can calculate the XPath for the relative attribute, the author.

To find the XPath that matches all book titles, we first select some sample books from the data set. We then search the DOM tree of the HTML document for titles of these samples. From this inspection, we will retrieve the XPaths to all elements containing title data from the samples. We then cluster, align and merge this information with the intent to extrapolate the XPaths from the samples into a single XPath that matches all book titles in the document, as is explained below.

Next, we turn our attention towards the relative attribute "author". For each sample of which the title was matched, we calculate the relative XPath to the nearest element containing author data. Under the assumption the HTML document is generated by a template, we expect this relative path will be the same for each book. However, we foresee the possibility some might differ for

unknown reasons. In order to retain only one XPath we use a majority voting system, choosing the relative path that is found the most.

Finally, we end up with a single XPath per attribute: an absolute one for the titles and a relative one for the authors. These can be interpreted as follows: the absolute XPath indicates where all book titles can be found in the document, and the relative XPath specifies where the author of a book can be found with respect to its title.

When comparing these XPaths to the ones we used for the initial configuration, we should find that they are equal.

6 DOM inspection

To retrieve the XPath to an element containing specific data, we crawl through the DOM tree recursively, inspecting each node we pass through. The algorithm is initialized at the root node, which is equivalent to the `html` element. At each call, we check if the node is a text node. In case it is, we verify whether its value matches the data we're searching for. In case it isn't, we descend deeper into the DOM tree by calling the function on each child of the current node.

Each time we can match the value of a text node to the data we're trying to locate, we add the XPath to that element to a list of results. At the end of the algorithm, we retrieve a list of XPaths to all elements that contain the data.

This algorithm depends on a function that performs text-based entity comparison. Its implementation has a big impact on the accuracy of the results. In its simplest form, this could be a case and diacritic insensitive string comparison, but for names of people this method should be extended to be more flexible. The exact implementation of this method justifies more research, as explained in 10. In our experiments, we have used a case and diacritic insensitive regex to match the wrapped data.

7 Key data

Given the XPaths to the key data we want to combine them in such a manner that we can use them to extrapolate where the rest of the records in the document can be found. Because the DOM tree inspection algorithm does not guarantee that only one XPath is returned per sample, we need to analyse the results before we can extrapolate them. If multiple XPaths are returned for a single sample, it means the data was found multiple times on the page. Likely at least one of the matches will be part of a list containing all the records we aim to extract. To identify which XPaths point to the elements in this list and which are false positives, we apply a clustering, alignment and merging technique. Because the clustering itself depends on alignment and merging, we explain it last.

7.1 Aligning and Merging

Aligning and merging a set of XPaths combines them into a single XPath. The resulting XPath has the characteristic that it matches at least all elements the

original XPaths match. Ideally, the merged XPath matches more elements, so it can be used to extrapolate information from some samples.

First, we align the XPaths according to an n-dimensional implementation of an adaptation to the Levenshtein algorithm to align lists [18]. The alignment will produce the optimal arrangement for all XPaths simultaneously resulting in the lowest merge cost.

Afterwards the XPaths can be merged into a minimal enclosing XPath by inserting wildcards for the differences in the aligned paths. Iterating over the items of the now equally long XPaths, we introduce a wildcard if at least one item differs from the others. A wildcard can be inserted for either tagname, index or both.

```
Unaligned:
    /html[1]/body[1]/div[1]/div[1]/ol[1]/li[1]/div[2]/h1[1]/a[1]
    /body[1]/div[1]/div[2]/ol[1]/li[1]/div[2]/h2[1]/a[1]
    /html[1]/body[1]/div[1]/div[3]/ol[1]/li[1]/div[2]/h3[1]

Aligned:
    /html[1]/body[1]/div[1]/div[1]/ol[1]/li[1]/div[2]/h1[1]/a[1]
            /body[1]/div[1]/div[2]/ol[1]/li[1]/div[2]/h2[1]/a[1]
    /html[1]/body[1]/div[1]/div[3]/ol[1]/li[1]/div[2]/h3[1]

Merged:
    */body[1]/div[1]/div/ol[1]/li[1]/div[2]/*[1]/*
```

Fig. 2. Example of aligning and merging several XPaths.

The example in Figure 2 shows how three similar XPaths are aligned and then merged. Through the alignment, the amount of wildcards to be inserted during merging is minimal. The merging step illustrates a wildcard insertion for index, tagname and both. An index wildcard is inserted by omitting the index specifier. A tagname wildcard is inserted by replacing the tag by an asterisk.

7.2 Clustering

The purpose of clustering is to identify groups of similar XPaths. Therefore we iteratively assign each XPath to a cluster based on an adaptation to the Levenshtein distance measure for lists [19]. XPaths are essentially ordered lists of items, identified by a tagname and index. Two items are considered equal if they have both the same tagname and index. The distance from an XPath to a cluster is calculated using the centroid of the cluster. The centroid is calculated by aligning and merging all XPaths in the cluster.

During the clustering, we assign each XPath to its closest cluster, where closest cluster is defined as the cluster with centroid with the lowest Levenshtein-list distance to the XPath to assign. If the closest cluster has a distance higher than a certain threshold or no such cluster is found, the XPath is placed into a new cluster.

At the end of the clustering, no XPaths in one cluster should have a distance to each other greater than the threshold. This means a cluster contains a set of

nodes that are similar. The following example shows how an XPath is assigned to a cluster.

```
XPath to assign:
    /body[1]/ul[2]/li[5]/div[1]/span[1]/span[3]/a[1]
  cluster 1: (distance = 1)
    /body[1]/ul[2]/li[1]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ul[2]/li[3]/div[1]/span[1]/span[3]/a[1]
  cluster 2: (distance = 5)
    /body[1]/div[1]/div[2]/div][4]/div[2]/a[1]/span[1]

After assignment:
  cluster 1:
    /body[1]/ul[2]/li[1]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ul[2]/li[3]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ul[2]/li[5]/div[1]/span[1]/span[3]/a[1]
  cluster 2:
    /body[1]/div[1]/div[2]/div][4]/div[2]/a[1]/span[1]
```

Fig. 3. An example showing how an XPath is assigned to a cluster.

Before we analyse relative data, we choose one cluster to continue with. Generally, the largest cluster is representative for the data we are searching. In case there are multiple clusters with the same size, a tie-breaker is needed. As a tie-breaker, we evaluate the centroids of all equally large clusters and count how many elements are matched. We continue by selecting the cluster that matches the most elements, assuming the bulk of the page contains data and the other clusters, matching less elements, contain outliers, which can often be found in a side menu. In case the amount of matched elements are equal again, a random cluster is selected. The selected cluster is called the dominant cluster.

8 Relative data

For each sample record for which we have an XPath in the dominant cluster, we verify if there are relative attributes. For each relative attribute, we use the DOM tree inspection algorithm to find the XPaths to where the data can be found on the document. We select the shortest relative XPath, which means the element in closest proximity of the key data, as each item in a relative path means navigating one hop in the DOM tree.

Ideally, and under the assumption of template based website generation, all relative paths for a relative attribute will be the same. However, we foresee some records may be divergent and choose for a majority voting system for each relative attribute to select the most represented relative path. Consider the following example where two books have the same relative path to their author and there is one outlier. The majority voting system will select the most frequent relative path as the XPath for the relative attribute author:

9 Experiments

To test our approach, we have set up two experiments. For each of them, we used the same data set which we retrieved from the Eci book store website [2]. From there, we have extracted over 1000 books using a data extraction tool based on manually configured XPaths. Each page in the domain contains information on 12 books.

First, we verify whether or not we can reconstruct the initially configured XPaths by testing our method on the same website. We discern two scenarios: one where we provide well-chosen sample books of which we know they can be found on the page, and one where we use the entire data set.

Second, we apply our approach on a different website [3], from De Standaard Boekhandel. We have manually verified this page contains books from the dataset, so we know this website can serve as an example of a changed HTML document, where the structure is altered but the data is the same. Each page in this domain contains information on 16 books.

In both experiments we measure how many books are automatically discovered using the reconstructed XPaths.

9.1 Unchanged HTML

Scenario 1: In the first scenario, we select well-chosen sample books to verify if our DOM tree inspection and clustering, merging and alignment techniques work properly. A well-chosen sample is a book of which we know in advance it can be found on the page. Because we test our approach on the same website we used to extract the data set from, we expect the reconstructed XPaths to be identical to the originals. We count the amount of books identified by the reconstructed XPaths and list the percentage of the expected books that are found.

Amount of samples	Books found	% found
1	1	8
2	12	100
4	12	100
8	12	100
12	12	100

Table 1. The amount of books found in the original document based on the amount of well-chosen samples used.

We identify that we need at least two samples to find all books. This is because we need more than one sample to extrapolate. Furthermore, these results

[2] `http://www.eci.be/boeken/roman-literatuur/literatuur`
[3] `http://www.standaardboekhandel.be/`

show that our approach produces the correct results. Moreover, no more than 2 well-chosen samples are needed to reconstruct the original XPaths with perfect accuracy.

Scenario 2: To simulate a more realistic scenario where it is not possible to identify well-chosen samples, we run the same experiment using the entire data set. We find the results are the same: all 12 books on the page are found and the original XPaths are reconstructed.

9.2 Changed HTML

We verify if our approach is able to construct XPaths to known data in an altered HTML document. We test our method using the entire data set on an alternate website, after manually verifying there are at least two books on the page that are also in our data set. We also evaluate the computed XPaths to verify all books are returned. In the results below, we have shortened some XPaths by omitting the part that is equal among all of them to better highlight their discrepancies.

```
Cluster 1 (dominant):
    /body[1]/ ... /ul[2]/li[1]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ ... /ul[2]/li[3]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ ... /ul[2]/li[5]/div[1]/span[1]/span[3]/a[1]
    /body[1]/ ... /ul[2]/li[11]/div[1]/span[1]/span[3]/a[1]
centroid: //body[1]/ ... /ul[2]/li/div[1]/span[1]/span[3]/a[1]

Cluster 2:
    /body[1]/div[1]/ ... /div[2]/div][4]/div[2]/a[1]/span[1]
    /body[1]/div[1]/ ... /div[2]/div][2]/div[2]/a[1]/span[1]
centroid: //body[1]/div[1]/ ... /div[2]/div/div[2]/a[1]/span[1]
```

Fig. 4. The results of the clustering, aligning and merging of the XPaths of matched book titles on an alternate HTML document.

We notice there are 16 books on the website. Of all samples, only 4 books were matched. We manually verify these are the only 4 books that are also present in the data set. Of those 4, 2 were found multiple times on the page. Indeed, there is a sidebox where bestsellers are listed and those 2 books are among them. However, the dominant cluster leads to the correct list of data on the website. Its centroid serves an the absolute XPath for all book titles. Furthermore, all matched books return the same calculated relative path for the author. Reconfiguring the data extraction tool with the newly constructed absolute XPath for the titles and relative XPath for the authors, we see all books are returned, as shown in Figure 5.

10 Future work

An important part of the DOM tree inspection is the function that does text-based entity comparison. The difficulty of the decision whether or not a text

Fig. 5. Part of the output of the automatically reconfigured data extraction tool, which now highlights the titles (green) and authors (yellow) of all books based on the reconstructed XPaths, without manual reconfiguration, for the website of De Standaard Boekhandel.

field contains information on an entity we're interested in should reflect the complexity of the human decision making process. Concluding if two strings of text represent the same data often results in a certain degree of hesitation. This is usually expressed in linguistic terms such as "probably" and "likely". To translate this into numbers we turn to soft computing, which allows us to model such terms using special functions. We can then add an uncertainty model to the results.

11 Conclusion

In this work, we have provided a way to extend XPath-based data extraction tools with a method to automatically (re)construct XPaths based on data. This creates the possibility to create smarter data extraction tools that use both structure and data to extract information. First, data is gathered from HTML document A and stored as a set of records, each consisting of a set of attributes, of which one is a mandatory key attribute. Second, this data is used to search HTML document B for nodes containing information, which are represented by their XPaths. These XPaths are then clustered, aligned and merged. The new XPaths then tell us where the information on records can be found in document B. Our approach succeeds in identifying the records by locating the key data and generates XPaths for all attributes with high accuracy.

References

1. Laender, A.H., Ribeiro-Neto, B.A., da Silva, A.S., Teixeira, J.S.: A brief survey of web data extraction tools. ACM Sigmod Record **31**(2) (2002) 84–93

2. Chang, C.H., Kayed, M., Girgis, M.R., Shaalan, K.F.: A survey of web information extraction systems. Knowledge and Data Engineering, IEEE Transactions on **18**(10) (2006) 1411–1428
3. Liu, W., Meng, X., Meng, W.: Vide: A vision-based approach for deep web data extraction. Knowledge and Data Engineering, IEEE Transactions on **22**(3) (2010) 447–460
4. Sugibuchi, T., Tanaka, Y.: Interactive web-wrapper construction for extracting relational information from web documents. In: Special Interest Tracks and Posters of the 14th International Conference on World Wide Web. WWW '05, New York, NY, USA, ACM (2005) 968–969
5. Myllymaki, J.: Effective web data extraction with standard xml technologies. Computer Networks **39**(5) (2002) 635–644
6. Pan, A., Raposo, J., Álvarez, M., Hidalgo, J., Viña, Á.: Semi-automatic wrapper generation for commercial web sources. In: Engineering Information Systems in the Internet Context. Springer (2002) 265–283
7. Buttler, D., Liu, L., Pu, C.: A fully automated object extraction system for the world wide web. In: Distributed Computing Systems, 2001. 21st International Conference on., IEEE (2001) 361–370
8. Crescenzi, V., Mecca, G., Merialdo, P., et al.: Roadrunner: Towards automatic data extraction from large web sites. In: VLDB. Volume 1. (2001) 109–118
9. Reis, D.d.C., Golgher, P.B., Silva, A., Laender, A.: Automatic web news extraction using tree edit distance. In: Proceedings of the 13th international conference on World Wide Web, ACM (2004) 502–511
10. Zhai, Y., Liu, B.: Web data extraction based on partial tree alignment. In: Proceedings of the 14th international conference on World Wide Web, ACM (2005) 76–85
11. Liu, L., Pu, C., Han, W.: Xwrap: An xml-enabled wrapper construction system for web information sources. In: Data Engineering, 2000. Proceedings. 16th International Conference on, IEEE (2000) 611–621
12. Zhu, J., Nie, Z., Wen, J.R., Zhang, B., Ma, W.Y.: Simultaneous record detection and attribute labeling in web data extraction. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2006) 494–503
13. Carlson, A., Schafer, C.: Bootstrapping information extraction from semi-structured web pages. Machine Learning and Knowledge Discovery in ... (2008)
14. Dalvi, N., Bohannon, P., Sha, F.: Robust web extraction: An approach based on a probabilistic tree-edit model. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. SIGMOD '09, New York, NY, USA, ACM (2009) 335–348
15. Dalvi, N., Kumar, R., Soliman, M.: Automatic wrappers for large scale web extraction. Proc. VLDB Endow. **4**(4) (January 2011) 219–230
16. Hao, Q., Cai, R., Pang, Y., Zhang, L.: From one tree to a forest: A unified solution for structured web data extraction. In: Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR '11, New York, NY, USA, ACM (2011) 775–784
17. Lerman, K., Minton, S., Knoblock, C.: Wrapper maintenance: A machine learning approach. J. Artif. Intell. Res.(JAIR) **18** (2003) 149–181
18. Dan Gusfield: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
19. Vladimir Levenshtein: Binary codes capable of correcting deletions, insertions and reversals. Physics Doklady **10**(8) (1966) 707–710