Cross-Supervised Synthesis of Web-Crawlers

Adi Omari
Technion
omari@cs.technion.ac.il

Sharon Shoham
Academic College of Tel Aviv Yaffo
sharon.shoham@gmail.com

Eran Yahav
Technion
yahave@cs.technion.ac.il

ABSTRACT

A web-crawler is a program that automatically and systematically tracks the links of a website and extracts information from its pages. Due to the different formats of websites, the crawling scheme for different sites can differ dramatically. Manually customizing a crawler for each specific site is time consuming and error-prone. Furthermore, because sites periodically change their format and presentation, crawling schemes have to be manually updated and adjusted. In this paper, we present a technique for automatic synthesis of web-crawlers from examples. The main idea is to use hand-crafted (possibly partial) crawlers for some websites as the basis for crawling other sites that contain the same kind of information. Technically, we use the data on one site to identify data on another site. We then use the identified data to learn the website structure and synthesize an appropriate extraction scheme. We iterate this process, as synthesized extraction schemes result in additional data to be used for re-learning the website structure. We implemented our approach and automatically synthesized 30 crawlers for websites from nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs, and hotels.

Categories and Subject Descriptors D.1.2 [Programming Techniques]: Automatic Programming; I.2.2 [Artificial Intelligence]: Program Synthesis

1 Introduction

A web-crawler is a program that automatically and systematically tracks the links of a website and extracts information from its pages. One of the challenges of modern crawlers is to extract *complex structured information* from different websites, where the information on each site may be represented and rendered in a different manner and where each data item may have multiple attributes.

For example, price comparison sites use custom crawlers for gathering information about products and their prices across the web. These crawlers have to extract the structured information describing products and their prices from sites with different formats and representations. The differences between sites often force a programmer to create a customized crawler for each site, a task that is time consuming and error-prone. Furthermore, websites may eventually change their format and presentation, therefore the crawling schemes have to be manually maintained and adjusted.

Goal The goal of this work is to automatically synthesize web-crawlers for a family of websites that contain the same kind of information but may significantly differ on layout and formatting. We assume that the programmer provides one or more hand-crafted web-crawlers for some of the sites in the family, and would like to automatically generate crawlers for other sites in the family. For example, given a family of four websites of online book stores (each containing tens of thousands of books), and a hand-crafted crawler for one of them, we automatically generate crawlers for the other three. Note that our goal is not only to extract data from web-sites, but to synthesize the programs that extract the data.

Existing Techniques Our work is related to *wrapper induction* [24]. The goal of wrapper induction is to automatically generate extraction rules for a website based on the regularity of pages inside the site. Our main idea is to try and leverage a similar regularity across multiple sites. However, because different sites may significantly differ on their layout, we have to capture this regularity at a more abstract level. Towards that end, we define an abstract *logical representation* of a website that allows us to identify commonality even in the face of different formatting details.

In contrast to supervised techniques [24, 25, 29, 5], which require labeled examples, and unsupervised techniques [2, 3, 8, 32, 31, 34, 40] that frequently require manual annotation of the extracted data, our approach uses *cross supervision*, where the learned extraction rules of one site are used to produce labeled examples for learning the extraction rules in another site.

Our technique uses XPath [7], a widely used web documents query language along with regular expressions. This makes our resulting extraction schemes human readable and easy to modify when needed. There has been some work on the problem of XPath *robustness* to site changes [9, 26, 28], trying to pick the most robust XPath query for extracting a particular piece of information. While robustness is a desirable property, our ability to efficiently synthesize a crawler circumvents this challenge as a crawler can be regenerated whenever a site changes.

Our Approach: Cross-Supervised Learning of Crawling Schemes We present a technique for automatically synthesizing data-extracting crawlers. Our technique is based on two observations: (i) sites with similar content have data overlaps, and (ii) in a given site, information with similar semantics is usually located in nodes with a similar location in the document tree.

Using these observations, we synthesize data-extracting crawlers for a group of sites sharing the same type of information. Starting from one or more hand-crafted crawlers which provide a relatively small initial set of crawled data, we use an iterative approach to discover data instances in new websites and extrapolate data ex-

ICSE '16, May 14-22, 2016, Austin, TX, USA

 ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884842

traction schemes which are in turn used to extract new data. We refer to this process as *cross-supervised learning*, as data from one web-site is repeatedly used to guide synthesis in other sites.

Our crawlers extract data describing different attributes of items. We introduce the notion of a *container* to maintain relationships between different attributes that refer to the same item. We use containers, which are automatically selected without any prior knowledge of the structure of the website, to handle pages with multiple items, and to filter out irrelevant data. This allows us to synthesize extraction schemes from positive examples only.

Our approach is scalable and practical: we used cross-supervision to synthesize crawlers for several product review websites, e.g., tvexp.com, weppir.com, camexp.com and phonesum.com.

Main Contributions The contributions of this paper are:

- A framework for automatic synthesis of web-crawlers. The main idea is to use hand-crafted crawlers for a number of websites as the basis for crawling other sites that contain the same kind of information.
- A new cross-supervised crawler synthesis algorithm that extrapolates crawling schemes from one web-site to another. The algorithm handles pages with multiple items and synthesizes crawlers using only positive examples.
- An implementation and evaluation of our approach, automatically synthesizing 30 crawlers for websites from nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs and hotels. The crawlers that we synthesize are real crawlers that were used to crawl more than $12,000$ webpages over all categories.

2 Overview

2.1 Motivating Example

Consider a price comparison service for books, which crawls book seller websites and provides a list of sellers and corresponding prices for each book. Examples of such book seller sites include barnesandnoble.com (B&N), blackwell.co.uk (BLACKWELL) and abebooks.com (ABE). Each of these sites lists a wide collection of books, typically presented in template generated webpages feeding from a database. Since these pages are template generated, they present *structured information* for each book in a format that is repeated across books. By recognizing this repetitive structure for a given site, one can synthesize a data extraction query and use it to automatically extract the entire book collection.

While the format within a single site is typically stable, the formats *between sites* differ considerably. Fig. 1 shows a small and simplified fragment of the page structure on B&N and BLACKWELL in HTML. Fig. 3 shows part of the tree representation of the corresponding sites (as well as of ABE), where $D_1, \ldots, D_4$ denote different pages. Due to the differences in structure, the data extraction query can differ dramatically. For example, in BLACKWELL and ABE, each of the pages (D_2, D_3, D_4) presents a single book, whereas in B&N the page D_1 shows a list of several books.

The goal of this work is to automatically synthesize crawlers for new sites based on some existing hand-crafted crawler(s). For example, given a crawler for the BLACKWELL site, our technique synthesizes a crawler for B&N website. The synthesized crawler is depicted in Fig. 2. We show that this can be done despite the significant differences between the sites BLACKWELL, and B&N, in terms of HTML structure. We note that the examples that we present in this section are abbreviated and simplified. For example, the real DOM tree for the B&N page we show here contains around $1,000$ nodes. The structure of the full trees, and the XPaths required for processing them are more involved than what is shown here.

barnesandnoble.com

```
<ol class="result-set box">
   <li class="result box">..
   <div class="details below-axis" >
    <a href="..." data-bntrack="Title_9781628718980"
      class="title" >
    THROUGH THE LOOKING GLASS</a>
    <a href=".."
    data-bntrack="Contributor_9781628718980"
      class="contributor" >
    David Winston Busch</a>
    ...
    <div class="price-format">
        <a href="..." data-bntrack="Paperback_Format">
              <span class="format">Paperback</span>
              <span class="price">$9.91</span>
        </a>
   </div>
   </div>
   ...</li>
   <li class="result box">..
   <div class="details below-axis" >
    <a href="..." data-bntrack="Title_9780071633604"
      class="title">
    Alice's Adventures in Wonderland</a>
    <a href=".." data-bntrack="Contributor_9780071633604"
    class="contributor" >Lewis Carroll</a>
    ...
    <div class="price-format">
      <a href="..." data-bntrack="Paperback_Format">
            <span class="format">Paperback</span>
            <span class="price">$6.49</span>
         </a>
   </div>
   </div>
   ...</li>
   </ol>
```

blackwell.com

```
<div id="product-biblio">
  <h1>Through the looking glass</h1>
  <a class="link_type1" href="/jsp/a/Lewis_Carroll">
    David Winston Busch
  </a>
  <div class="price-info" align="center">
    <span class="price">
    £8.99</span>
   </div>
</div>
```

Figure 1: Fragments of webpages with the similar attribute values for a book on two different book shopping sites.

2.2 Cross-Supervised Learning of Crawling Schemes

Our main observation is that despite the significant differences in the concrete layout of websites, the pages of websites that exhibit the same product category often share the same *logical structure*; they present similar attributes for each product. For example, each of the pages of Fig. 1 presents the same important attributes about the book, including its *title*, *author name* and *price*. Moreover, there is a large number of shared products between these websites. The book *"Through the looking glass"* is one such example for B&N and BLACKWELL.

Our technique exploits data overlaps across sites in order to learn the concrete structure of a new website s based on other websites. Specifically, we identify in s concrete data extracted from other sites and as such learn the structure in which this data is represented in s. We then use multiple examples of the structure in which the data appears in s in order to generalize and get an extraction query for s. This enables our algorithm to extrapolate a crawler for s.

We do not require a precise match of data across sites, as our technique also handles noisy data. (For example, prices do not have

```
class MySpider(CrawlSpider):
 name = "barnesandnoble"
 allowed_domains = ["www.barnesandnoble.com"]
 start_urls = [
    ("http://www.barnesandnoble.com/s/java-programming
?store=allproducts&keyword=java+programming")
    ]

 rules = (
  Rule(LinkExtractor(
       allow=("/s/.*"),callback="parse_item", follow=True
  ),
 )

 def parse_item(self, response):
  sel = Selector(response)
  rows = sel.XPath('//body/div/div/section/div/ol["result-set
box"]/li[@class="result box"]/div/div[@class="details
below-axis"]')
  for r in rows:
   item = BooksItem()
   item['title'] = r.XPath(
     '//a[@class="title"]'
   ).extract()
   item['author'] = r.XPath(
     '//a[@class="contributor"]'
   ).extract()
   item['price'] = r.XPath(
     '//div[@class="price-format"]/a/span[@class="price"]'
   ).extract()
   yield item
```

Figure 2: Crawler for java books from Barnes&Noble.

to be identical; any number can be a match.)

Crawling schemes A crawler, such as the one of Fig. 2, contains some boilerplate code defining the crawler class and its operations. However, the essence of the crawler is its *crawling scheme*. For example, in Fig. 2 the crawling scheme is highlighted in boldface.

A crawling scheme is defined with respect to a set of semantic groups, called *attributes*, which define the types of data to be extracted. In the books example, the attributes are: *book title*, *author* and *price*.

Given a set of attributes, a crawling scheme consists of the following two components: (i) A data extraction query that defines how to obtain values of the attributes for each item listed on the site. (ii) A starting point URL and a URL filtering pattern which let the crawler locate "relevant" pages and filter out irrelevant pages without downloading and processing them.

Our crawlers use XPath as a query language for data extraction. XPath is a query language for selecting nodes from an XML document which is based on the tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by describing their path from the document tree root node. For example, Fig. 4 and Fig. 5 show the crawling schemes for crawling books from BLACKWELL and B&N respectively, where the data extraction query is expressed using XPaths.

Two-level data extraction schemes We assume that the data extraction query has two levels: The first level query is an XPath describing an item container. Intuitively, a container is a sub-tree that contains all the attribute values we would like to extract (defined more formally in Sec. 5.) For example, in Fig. 4, the XPath `//body/div[@class="content__maincore-shop"]...` describes a container of book attributes on BLACKWELL pages.

The second level queries contain an extraction XPath for values of each individual attribute. These XPaths are relative to the root of the container. For example, `//div/h1/` in Fig. 4 is used to pick the node that has type `h1` (heading 1), containing the book title.

Iterative synthesis of crawling schemes Our approach considers a set of websites, and a set of attributes to extract. To bootstrap the synthesis process, the user is required to provide the set of websites for which crawler synthesis is desired, as well as a crawling scheme for at least one of these sites. Alternatively, the user can provide multiple partial crawling schemes for different sites, that together cover all the different item attributes.

The synthesis process starts by invoking the provided extraction scheme(s) on the corresponding sites to obtain an initial set of values for each one of the attributes. These values are then used to locate nodes that contain attribute values in the document trees of webpages of new sites. The nodes that contain attribute values reveal the structure of pages of the corresponding websites. In particular, smallest subtrees that exhibit all the attributes amount to containers. This allows for synthesis of data extraction schemes for new websites. The newly learned extraction schemes are used to extract more values and add them to the set of values of each attribute, possibly allowing for additional websites to be handled. This process is repeated until complete extraction schemes are obtained for all websites, or until no additional values are extracted.

In our example, the algorithm starts with the data extraction scheme for BLACKWELL (see Fig. 4), provided by a user. It extracts from D_2 `author-x`, `title-x`, and `price` as values of the book title, author, and price attributes, respectively (see Fig. 3). These values are identified in D_1 (B&N) within the subtree of the left most node represented by

```
//body/.../ol["result-set box"]
  /li[@class="result box"]/...
  /div[@class="details below-axis"],
```

which then points to the latter node as a possible container. Additional values taken from D_3 and other pages in BLACKWELL identify additional nodes in the B&N tree as attribute and container nodes. Note that `author-x` is also found in another subtree in D_1. However, there are no instances of the remaining attributes in that subtree; Therefore, the subtree is not considered a container and the corresponding node is treated as noise.

By identifying the commonality between the identified containers and between nodes of the same attribute, a data extraction scheme for B&N is synthesized (see below). In the next iteration, the new data scheme is used to extract from B&N the values `author-z`, `title-z` and `price` as additional values for book title, author, and price respectively (that did not exist in BLACKWELL). The new values are located in ABE (see D_4 in Fig. 3), allowing to learn an extraction scheme for ABE as well.

XPath synthesis for two-level queries Our approach synthesizes a two level extraction scheme for each website from a set of attribute nodes and candidate containers identified in its webpages. The two-level query structure is reflected also in the synthesis process of the extraction scheme. Technically, we use a two-phase approach to synthesize the extraction scheme. In each site, we first generate an XPath query for the containers. We then filter the attribute nodes keeping only those reachable from containers that agree with the container XPath, and generate XPaths for their extraction relatively to the container nodes.

To generate an XPath query for a set of nodes (e.g., for the set of containers), we consider the concrete XPath of each node—this is an XPath that extracts exactly this node. We unify these concrete XPaths by a greedy algorithm that aims to find the most concrete (most strict) XPath query that agrees with a majority of the concrete XPaths. Keeping the unified XPath as concrete as possible prevents the addition of noise to the extraction scheme.

The generated XPaths for B&N are depicted in Fig. 5. In this example, unification is trivial since the XPaths are identical. How-

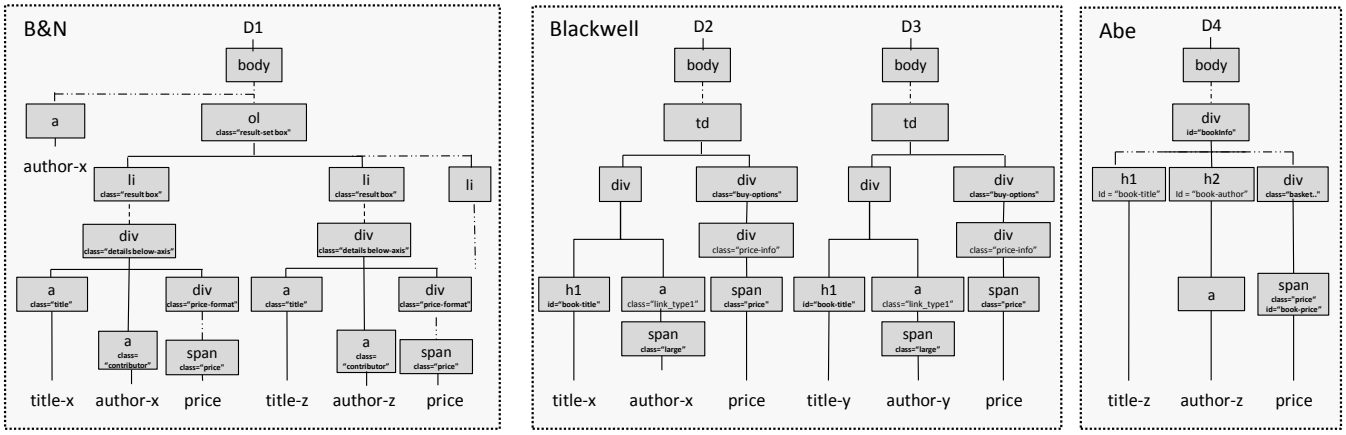

Figure 3: Example DOM trees

Container:	`//body/div[@class="content__maincore-shop"]`
	`/table[@class="main-page"]/tr/`
	`td[@class="two-col-right"]/table/tr/td`

Title:	`//div/h1/`
Author:	`//div/a[@class="link_type1"]`
Price:	`//div[@id="buy-options"]/div/span`
URL Pattern:	`.*jsp/id/.*`

Figure 4: Crawling scheme for BLACKWELL.

Container:	`//body/div/div/section/div/ol["result-set box"]`
	`/li[@class="result box"]/div`
	`/div[@class="details below-axis"]`

Title:	`//a[@class="title"]`
Author:	`//a[@class="contributor"]`
Price:	`//div[@class="price-format"]` `/a/span[@class="price"]`
URL Pattern:	`/s/.*`

Figure 5: Crawling scheme for B&N.

ever, if for example each of the container nodes labeled `div` in D_1 had different `id`'s, the `id` feature would have been removed during unification. Note that even if the subtree that contains the noisy instance of `author-x` in D_1 had been identified as a candidate container (e.g., if it had contained values of the other attributes), it would have been discarded during the unification.

URL pattern synthesis In order to synthesize a URL pattern for the crawling scheme of a new site, we extend the iterative technique used for synthesis of data extraction schemes; In each iteration of the algorithm, for each website we identify a set of pages of interest as pages that contain attribute data. We filter these pages in accordance with the filtering of container and attribute nodes. We then unify the URLs of remaining pages similarly to XPath unification.

Fig. 5 depicts the URL pattern generated by our approach for B&N. This pattern identifies webpages in B&N that present a list of books—these are the pages whose structure conforms with the synthesized extraction scheme. Note that B&N also presents the same books in a separate page each, but such pages require a different crawling scheme.

3 Preliminaries

In this section we define some terms that will later be used to describe our approach.

3.1 Logical Structure of Webpages

Each webpage implements some *logical structure*. Following [19], we use *relations* as a logical description of data which is independent of its concrete representation. A relational specification is a set of relations, where each relation is defined by a set of column names and a set of values for the columns. A *tuple* $t = \langle c_1 : d_1, c_2 : d_2, \ldots \rangle$ maps a set of columns $\{c_1, c_2, \ldots\}$ to values. A *relation* r is a set of tuples $\{t_1, t_2, \ldots\}$ such that the columns of every $t, t' \in r$ are the same.

For example, B&N, BLACKWELL and ABE described in Section 2 implement a relational description of a list of books, where each book has a title, an author and a price. Then "book title", "author" and "price" are columns, and the set of books is modeled as a relation with these columns, where each tuple is a book item.

Data items, attributes and instances We refer to each tuple of a relation r as a *data item*. The columns of a relation r are called *attributes*, denoted *Att*. Each attribute defines a class of data sharing semantic similarities, such as meaning and/or extraction scheme. The value of attribute $a \in Att$ in some tuple of r is also called an *instance* of a. The set of all values of all attributes is denoted V. Each attribute a is associated with an *equivalence relation* $\equiv_a$ that determines if two values are equivalent or not as instances of a. (The notion of "equivalence" may differ between different attributes.) By default (if not specified by the user) we use the bag of words representation of each value d, denoted $W(d)$, and use Jaccard similarity function [21], $J(d_1, d_2)$, with a threshold of 0.5 as an equivalence indicator between values d_1 and d_2:

$$d_1 \equiv_a d_2 \text{ iff } J(d_1, d_2) > 0.5 \text{ where } J(d_1, d_2) = \frac{|W(d_1) \cap W(d_2)|}{|W(d_1) \cup W(d_2)|}.$$

3.2 Concrete Layout of Webpages

Technically, webpages are documents with structured data, such as XML or HTML documents. The concrete layout of the webpage implements its logical structure, where attribute instances are presented as nodes in the DOM tree.

XML documents as DOM trees A well formed XML document, describing a webpage of some website, can be represented by a DOM tree. A DOM tree is a labeled ordered tree with a set of nodes N and a labeling function that labels each node with a set of node features (not to be confused with data attributes), where some of the features might be unspecified. Common node features include `tag`, `class` and `id`.

For example, Fig. 3 depicts part of the tree representation of pages of B&N, BLACKWELL and ABE. A node labeled by `a`, `class=title` is a node whose `tag` is `a`, `class` is `title`, and `id` is unspecified.

Node descriptors A *node descriptor* is an expression x in some language defining a set of nodes in the DOM tree. We use *Expr* to denote the set of node descriptors. For a node descriptor $x \in$ *Expr* and a webpage p, we define $[\![x]\!]_p$ to be the set of nodes described by x from p. When p is clear from the context, we omit it from the notation. A node descriptor is *concrete* if it represents exactly one node. We sometimes also refer to node descriptors as *extraction schemes*. In this work, we use XPath as a specification language for node descriptors.

3.3 XPath as a Data Extraction Language

XPath [7] is a query language for traversing XML documents. XPath expressions (XPaths in short) are used to select nodes from the DOM tree representation of an XML document. An XPath expression is a sequence of instructions, $x = x_1 \dots x_k$. Each instruction x_i defines how to obtain the next set of nodes given the set of nodes selected by the prefix $x_1 \dots x_{i-1}$, where the empty sequence selects the root node only. Roughly speaking, each instruction x_i consists of (i) *axis* defining where to look relatively to the current nodes: at children ("/"), descendants ("//"), parent, siblings, (ii) *node filters* describing which `tag` to look for (these can be "all", "text", "comment", etc.), and (iii) *predicates* that can restrict the selected nodes further, for example by referring to values of additional node features (e.g. `class`) that should be matched.

For example, the XPath `//div/*/a[@class="link_type1"]` selects all nodes that follow a sequence of nodes that can start anywhere in the DOM tree, and has to consist of a node with `tag=div` followed by some node whose features are unspecified and is followed by a node with `tag=a` and `class=link_type1`.

4 The Crawler Synthesis Problem

In this section we formulate the crawler synthesis problem. A crawler for a website can be divided into two parts: a *page crawler*, and a *data extractor*. The page crawler is responsible for grabbing the pages of the site that contain relevant information. The data extractor is responsible for extracting data of interest from each page.

Logical structure of interest Our work considers websites whose data-containing webpages share the following logical structure: each webpage describes one main relation, denoted *data*. As such, data items are tuples of the *data* relation. Further, the set *Att* of attributes consists of the columns of the *data* relation.

Note that different concrete layouts can implement this simple logical structure. For example, if we consider a webpage that exhibits a list of books, then the concrete layout can first group books by author, and for each author list the books, or it can avoid the partition based on authors. Further, some websites will present each book in a separate webpage, whereas others will list several books in the same page. Even for websites that are structured similarly by the former parameters, the mapping of attribute instances to nodes in the DOM tree can vary significantly.

Page crawlers A page crawler for a website s is given by a URL pattern, denoted $U(s)$, which identifies the set of webpages of interest. These are the webpages of the website that contain data of the relevant kind. We denote by $P(s)$ the set of webpages whose URL matches $U(s)$.

Data Extractors Recall that we consider webpages where instances of different attributes are grouped into tuples of some relation, denoted *data*. We are guided by the observation that data in such webpages is typically stored in subtrees, where each subtree contains instances of all attributes for some data item (i.e., tuple of the *data* relation). We refer to the roots of such subtrees as *containers*:

Containers: A node in the DOM tree whose subtree contains all the entries of a single data item (i.e., a single tuple of *data*) is called a *container*. Note that any ancestor of a container is also a container. We therefore also define the notion of a *best* container to be a container such that none of its predecessors is a container. Depending on the concrete layout of the webpage, a best container might correspond to an element in a list or in another data structure. It might also be the root of a webpage, if each webpage presents only one data item.

For example, in the tree D_1 depicted in Fig. 3, both of the nodes selected by `//body/.../div[@class="details below-axis"]` are containers, and as such so are their ancestors, including the root. However, the latter are not best containers since they include strict subtrees that are also containers.

Attribute nodes: A node in the DOM tree that holds an instance of an attribute $a \in$ *Att* is called an *a-attribute node*, or simply an *attribute node* when a is clear from the context or does not matter.

Data extractors: A data extractor for the relation *data* over columns *Att* in some website s can be described by a pair $(container, f)$ where $container \in$ *Expr* is a node descriptor representing containers, and f : *Att* $\hookrightarrow$ *Expr* is a possibly partial function that maps each attribute name to a node descriptor, with the meaning that this descriptor represents the attribute nodes relatively to the container node, i.e., the attribute descriptor considers the container node as the root. The data extractor is *partial* if f is partial. If $container$ is empty, it is interpreted as a node descriptor that extracts the root of the page. If $container$ is empty and f is undefined for every attribute, we say that the data extractor is *empty*.

Examples of data extractors appear in Fig. 5 and Fig. 4.

Crawler synthesis The *crawler synthesis problem* w.r.t. a set *Att* of attributes is defined as follows. Its input is a set S of websites, where each website $s \in S$ is associated with a data extractor, denoted $E(s)$, over *Att*. $E(s)$ might be partial or even empty. The desired output is a page crawler, along with a complete data extractor for every $s \in S$.

5 Data Extractor Synthesis

In this section we focus on synthesizing data extractors, as a first step towards synthesizing crawlers. We temporarily assume that the page crawler is given, i.e., for each website we have the set of webpages of interest, and present our approach for synthesizing data extractors. We will remove this assumption later, and also address synthesis of the page crawler, using similar techniques.

The input to the data extractor synthesis is therefore a set S of websites, where each website $s \in S$ is associated with a set of webpages, denoted $P(s)$, and with a data extractor, denoted $E(s)$, which might be partial or even empty. The goal is to synthesize a complete data extractor for every $s \in S$. The main challenge in synthesizing a data extractor is identifying the mapping between the logical structure of a webpage, and its concrete layout as a DOM tree. The key to understanding this mapping amounts to identifying the container nodes in the DOM tree that contain all the attributes of a single data item (tuple). Once this mapping is learnt, the next step is to capture it by synthesizing extraction schemes in the form of XPaths.

The data extractor synthesis algorithm is first described using the generic notion of node descriptors. In Section 5.2 we then instantiate it for the case where node descriptors are provided by XPaths.

Before we describe our algorithm, we review its main ingredi-

ents. In the following, we use $N(p)$ to denote the set of nodes in the DOM tree of a webpage $p \in P(s)$.

Knowledge base of data across websites Our synthesizer maintains a knowledge base $O : \mathit{Att} \rightarrow 2^V$ which consists of a set of observed instances for each attribute $a \in \mathit{Att}$. These are instances collected across different websites from S. They enable the synthesizer to locate potential a-attribute nodes in webpages for which the data extractor of a is unspecified.

Data to node mapping per website In addition to the global knowledge base, for each website $s \in S$ our synthesizer maintains: (i) a set $N^{\mathit{cont}}(p) \subseteq N(p)$ of (candidate) container nodes for each webpage $p \in P(s)$, and (ii) a set $N^a(p) \subseteq N(p)$ of (candidate) attribute nodes for each webpage $p \in P(s)$ and attribute $a \in \mathit{Att}$.

Deriving extraction schemes per website The synthesis algorithm iteratively updates the container and attribute node sets for each webpage in $P(s)$, and attempts to generate a data extractor $E(s)$: $\mathit{Expr} \times (\mathit{Att} \hookrightarrow \mathit{Expr})$ for s by generating node descriptors for the set of containers, and for each of the attributes. The extraction scheme is shared by all webpages of the website. The updates of the sets and the attempts to generate node descriptors from the sets are interleaved, as one can affect the other; on the one hand node descriptors are generated in an attempt to represent the sets; on the other hand, once descriptors are generated, elements of the sets that do not conform to them are removed.

While attribute instances are used to identify attribute nodes across different websites, the synthesis of node descriptors is performed for each website separately and independently of others (while considering all of the webpages associated with the website).

5.1 Algorithm

Algorithm 1 presents our data extractor synthesis algorithm. The algorithm is iterative, where each iteration consists of two phases:

Phase 1: Data extraction for knowledge base extension. Initially, the sets $O(a)$ of instances of all attributes $a \in \mathit{Att}$ are empty. In each iteration, we use yet un-crawled extraction schemes to extract attribute nodes in all webpages of all websites and extend the sets $O(a)$ for every attribute a based on the content of the extracted nodes. At the first iteration, input extraction schemes are used. In later iterations, we use newly learnt extraction schemes, generated in phase 2 of the previous iteration.

Phase 2: Synthesis of data extractors. For every website $s \in S$ for which the extraction scheme is not yet satisfactory, we attempt to generate an extraction scheme by performing the following steps:

(1) *Locating attribute nodes per page:* We traverse all webpages $p \in P(s)$ and for each attribute a we use the instances $O(a)$ collected in phase 1 (from this iteration and previous ones) to identify potential a-attribute nodes in p. Technically, for every $p \in P(s)$ we iterate on all $n \in N(p)$ and use the (default or user-specified) equivalence relation $\equiv_a$ to decide whether n contains data that matches the attribute instances in $O(a)$. If so, n is added to $N^a(p)$.

(2) *Locating container nodes per page:* In every webpage $p \in P(s)$ we locate potential container nodes, and collect them in $N^{\mathit{cont}}(p)$. A container is expected to contain instances of all attributes *Att*. However, since our knowledge of the attribute instances is incomplete, we need to also consider subsets of *Att*. In each webpage, we define the "best" set of attributes to be the set of all attributes whose instances appear in it. Potential containers are nodes whose subtree contains attribute nodes of the "best" set of attributes, and no strict subtree contains nodes of the same set of attributes. The latter ensures that the container is best. Technically, for every node $n \in N(p)$ we compute the set of reachable attributes $a \in \mathit{Att}$ such that an a-attribute node in $N^a(p)$ is reachable from n. Nodes n whose set is best and no other node reachable from n has the same set of reachable attributes are collected in $N^{\mathit{cont}}(p)$. For each container node $n_c \in N^{\mathit{cont}}(p)$ we also maintain its *support* - the number of attribute nodes reachable from it.

(3) *Generating container descriptor:* We consider the concrete node descriptor of every container node $n_c \in N^{\mathit{cont}}(p)$ in every webpage $p \in P(s)$. We unify the concrete node descriptors across all webpages into a single node descriptor, and use it to update $E(s)$, relying on the observation that containers are typically elements of some data structure and are therefore accessed similarly.

(4) *Filtering attribute nodes based on container descriptor:* We filter the sets $N^{\mathit{cont}}(p)$ of containers in all webpages to keep only containers that match the unified node descriptor, and accordingly filter the sets $N^a(p)$ of attribute nodes in all webpages to contain only nodes that are reachable from the filtered sets of containers. This step enables us to automatically distinguish the nodes we are interested in from others that accidentally contain attribute instances, without any a-priori knowledge.

(5) *Generating attribute descriptors:* For each attribute $a \in \mathit{Att}$, we consider the concrete node descriptors of all the nodes in the filtered sets $N^a(p)$ of all webpages $p \in P(s)$, where the concrete node descriptor of n is computed relatively to the container node whose subtree contains n. For each attribute a, we find a unified node descriptor for these concrete node descriptors, and use it to update $E(s)$. Again, we use the observation that containers are structured similarly and therefore attribute data within them is accessed similarly.

Remark. For a successful application of our algorithm, at least one extraction scheme should be provided for every attribute. Our approach is also applicable if a user provides a set of annotated webpages instead of a set of initial extraction expressions.

Section 2 describes a running example of our algorithm.

Node descriptor unification Node descriptors for the container and attributes are generated by unifying concrete node descriptors of the nodes in $N^{\mathit{cont}}(p)$ and $N^a(p)$ respectively. Roughly speaking, the purpose of the unification is to derive a node descriptor that is general enough to describe as many of the concrete node descriptors as possible, but also as concrete as possible in order to introduce as little noise as possible. "Concreteness" of a node descriptor x is measured by an *abstraction score*, denoted $\mathrm{abs}(x)$. The node descriptor unification algorithm is parametric in the abstraction score. In Section 5.2, we provide a definition of this score when the node descriptors are given by XPaths.

DEFINITION 5.1. *For a set X of concrete node descriptors and a weight function support that associates each $x \in X$ with its support, the* unification problem *aims to find a node descriptor x_g, s.t.:*

1. $\frac{\mathit{support}(\{x \in X \mid [\![x]\!] \subseteq [\![x_g]\!]\})}{\mathit{support}(X)} > \delta$*, i.e., x_g captures at least δ of the total support of the node descriptors in X.*
2. $\mathit{abs}(x_g)$ *is minimal.*

In container descriptor unification (step 3), the given node descriptors represent container nodes. The support of each descriptor represents the number of attribute nodes reachable from the container. In attribute descriptors unification (step 5), the given descriptors represent attribute nodes for some attribute, all of which are reachable from a set of containers of interest. The attribute node descriptors are relative to the container nodes.

5.2 Implementation using sequential XPaths

In order to complete the description of our data extractor synthesizer, we describe how the ingredients of Algorithm 1 are implemented when node descriptors are given by XPaths. Specifically, our approach uses sequential XPaths:

Algorithm 1: Data Extractor Synthesizer

Input: set of attributes *Att*
Input: set of websites S
Input: a map $E : S \to (Expr \times (Att \hookrightarrow Expr))$ mapping a website s to a data extractor $E(s)$ which consists of a (possibly empty) container descriptor as well as a (possibly partial) mapping of attributes to node descriptors

$O = []$
while *there is change in* O *or* E **do**
  `/* Data extraction phase */`
  foreach $s \in S$ *s.t.* $E(s)$ *is uncrawled* **do**
    $O = O \cup$ ExtractInstances $(Att, P(s), E(s), O)$
  `/* Synthesis phase */`
  foreach $s \in S$ *s.t.* $E(s)$ *is incomplete* **do**
    `/* Locate attribute nodes */`
    foreach $p \in P(s)$ **do**
      foreach $a \in Att$ **do**
        $N^a(p) =$ FindAttNodes $(N(p), a, O(a))$
    `/* Locate container nodes */`
    foreach $p \in P(s)$ **do**
      $bestAttSet = \{a \in Att \mid N^a(p) \neq \emptyset\}$
      foreach $n \in N(p)$ **do**
        $reachAtt[p][n] = \{a \in Att \mid \exists n' \in reach(n) : n' \in N^a(p)\}$
        $support[p][n] = \#\{n' \in reach(n) \mid \exists a \in Att : n' \in N^a(p)\}$
      $N^{cont}(p) = candidates = \{n \in N(p) \mid reachAtt[n] = bestAttSet\}$
      foreach $n \in candidates$ **do**
        foreach $n' \in children(n)$ **do**
          if $n' \in candidates$ **then**
            $N^{cont}(p) = N^{cont}(p) \setminus \{n\}$
            break
    `/* Generate container descriptor */`
    $Exprs =$
     $\{(relativeExpr(p, emptyExpr, n), support[p][n]) \mid p \in P(s), n \in N^{cont}(p)\}$
    $containerExpr =$ UnifyExpr $(Exprs)$
    FilterAttributeNodes()
    `/* Generate attribute descriptors */`
    foreach $a \in Att$ **do**
      $Exprs =$
       $\{(relativeExpr(p, containerExpr, n), 1) \mid p \in P(s), n \in N^a(p)\}$
      $attExpr[a] =$ UnifyExpr $(Exprs)$
    $E(s) = (containerExpr, attExpr)$
return E

Sequential XPaths A *path* π in the DOM tree is a sequence of nodes $n_1, \ldots, n_k$, where for every $1 \leq i < k$, there is an edge from n_i to n_{i+1}. Such a path can naturally be encoded using an XPath $\texttt{XS}(\pi) = x_1 \ldots x_k$ where each x_i starts with "/". x_1 may start with "//" rather than "/" if π does not necessarily start at the root of the tree. Further, each x_i uses node filters and predicates to describe the features of n_i. Therefore, x_i can be described via equalities $f_1 = v_1, \ldots, f_m = v_m$, such that $f_j \in F$, where F is the set of node features used. We consider $F = \{\texttt{tag}, \texttt{class}, \texttt{id}\}$ for simplicity, but our approach is not limited to these features. A feature might be unspecified for n_i, in which case no corresponding equality will be included in x_i.

For example, let π be the left most path in D_2 (Fig. 3). Then $\texttt{XS}(\pi) =$ `//body/.../td/div/h1`. $\texttt{XS}(\pi)$ can also be described as a sequence $\langle$`tag=body`$\rangle \ldots \langle$`tag=td`$\rangle\langle$`tag=div`$\rangle\langle$`tag=h1`$\rangle$.

We refer to XPaths of the above form as *sequential*. The XPaths that our approach generates as node descriptors are all sequential.

Concrete XPaths Each node n in the DOM tree can be uniquely described by the unique path, denoted π_n, leading from the root to n. The XPath $\texttt{XS}(\pi_n)$ is a sequential XPath such that $[\![\texttt{XS}(\pi_n)]\!] \supseteq \{n\}$, and $[\![\texttt{XS}(\pi_n)]\!]$ is minimal (i.e., every other sequential XPath that also describes n, describes a superset of $[\![\texttt{XS}(\pi_n)]\!]$). We therefore refer to $\texttt{XS}(\pi_n)$ as the *concrete* XPath of n, denoted $\texttt{XS}(n)$ with abuse of notation. (If we include in F the position of a node among its siblings as an additional node feature, and encode it by an XPath instruction using sibling predicates then we will have $[\![\texttt{XS}(\pi_n)]\!] = \{n\}$).

Agreement of sequential XPaths We observe that for sequential XPaths, checking if a node n matches a node descriptor x_g (i.e. $n \in [\![x_g]\!]$) can be done by checking if the concrete XPath $\texttt{XS}(n)$ *agrees* with the XPath x_g, where agreement is defined as follows.

DEFINITION 5.2. *Let* $x = x_1 \ldots x_k$ *and* $x_g = x_1^g \ldots x_m^g$ *be sequential XPaths. The instruction* x_i agrees *with instruction* x_i^g *if whenever some feature is specified in* x_i*, it either has the same value in* x_i^g *or it is unspecified in* x_i^g*. The XPath* x agrees *with the XPath* x_g *if* $m \leq k$*, and for every* $i \leq m$*,* x_i *agrees with* x_i^g*.*

For example, `//body/.../td/div[id=name1]/h1` agrees with both `//body/.../td/div/h1`, and `//body/.../td/div`.

Node descriptor unification via XPath unification We now describe our solution to the node descriptor unification problem in the setting of sequential XPaths. We first define the abstraction score:

Abstraction score For a sequential XPath instruction x_i we define $spec(x_i)$ to be the subset of features whose value is specified in x_i, and $unspec(x_i) = F \setminus spec(x_i)$ is the set of unspecified features in x_i. We define the abstraction score of x_i to be the number of features in $unspec(x_i)$, that is, $\text{abs}(x_i) = |unspec(x_i)|$.

For a sequential XPath $x = x_1 \ldots x_k$, we define $\text{abs}(x)$ to be the sum of $\text{abs}(x_i)$.

Greedy algorithm for unification Algorithm 2 presents our unification algorithm. We use the observation that for sequential XPaths, the condition $[\![x]\!] \subseteq [\![x_g]\!]$ that appears in item 1 of the unification problem (see Definition 5.1) can be reduced to checking if the XPath x *agrees* with the XPath x_g.

Let X be a weighted set of sequential XPaths, with a weight function *support* that associates each XPath in X with its support. Let $TS = support(X)$ denote the total support of XPaths in X. The unification algorithm selects k to be the length of the longest XPath in X. It then constructs a unified XPath $x_g = x_1^g, \ldots, x_m^g$ top down, from $i = 1$ to k (possibly stopping at $i = m < k$). Intuitively, in each step the algorithm tries to select the most "concrete" instruction whose support is high enough. Note that there is a tradeoff between the high-support requirement and the high-concreteness requirement. We use the threshold as a way to balance these measures.

At iteration i of the algorithm, X^{i-1} is the restriction of X to the XPaths whose prefix agrees with the prefix $x_1^g, \ldots, x_{i-1}^g$ of x_g computed so far (Initially, $X^0 = X$). We inspect the i'th instructions of all XPaths in X^{i-1}. The corresponding set of instructions is denoted by $I^i = \{x_i \mid x \in X^{i-1}\}$. The support of an instruction x_B w.r.t. I^i is $support(\{x \in X^{i-1} \mid x_i \text{ agrees with } x_B\})$.

To select the most "concrete" instruction whose support is high enough, we consider a predefined order on sets of feature-value pairs, where sets that are considered more "concrete" (i.e., more "specified") precede sets considered more "abstract". Technically, we consider only feature-value sets where each feature has a unique value. The order on such sets used in the algorithm is defined such that if $|B_1| > |B_2|$ then B_1 precedes B_2. In particular, we make sure that sets where all features are specified are first in that order.

For every set B of feature-value pairs, ordered by the predefined order, we consider the instruction x_B that is specified exactly on

the features in B, as defined by B. If its support exceeds δ, we set x_i^g to x_B and X^i to $\{x \in X^{i-1} \mid x_i \text{ agrees with } x_B\}$. Otherwise, x_B is not yet satisfactory and the search continues with the next B. There is always a B for which the support of the x_B exceeds the threshold, for instance, the last set B is always the empty set with $x_B = /*$, which agrees with all the concrete XPaths in X^{i-1} .

If at some iteration $I^i = \emptyset$, i.e. the XPaths in X^{i-1} are all of length $< i$ and therefore there is no "next" instruction to discover, the algorithm terminates. Otherwise, it terminates when $i = k$.

EXAMPLE 1. *Given the following concrete XPaths as an input:*

$cx_1 =$ `/div[class="title"]/span/a[id="t1"]`
$cx_2 =$ `/div[class="title"]/span/a[id="t2"]`
$cx_3 =$ `/div[class="note"]/span/a[id="n1"]`

The unification starts with $X^0 = \{cx_1, cx_2, cx_3\}$, and $i = 1$. To select x_1^g, recall that the algorithm first considers the most specific feature-value sets (in order to find the most specific instruction). In our example it starts from $B_1 =$ {`tag=div`, `class=note`} for which $x_{B_1} =$ `/div[class="note"]`. However, cx_3 is the only XPath in X^0 which agrees with x_{B_1}. Therefore it has support of $1/3$. We use a threshold of $\delta = 1/2$. Thus, the support of x_{B_1} is insufficient. The algorithm skips to the next option, obtaining $x_{B_2} =$ `/div[class="title"]`. This instruction is as specific as x_{B_1} and has a sufficient support of 2/3 (it agrees with cx_1 and cx_2). Therefore, for $i = 1$, the algorithm selects $x_1^g = x_{B_2}$ and $X^1 = \{cx_1, cx_2\}$. For $i = 2$, the algorithm selects $x_2^g =$ `/span` as the most specific instruction, which also has support of 2/2 (both cx_1 and cx_3 from X^1 agree with it). For $i = 3$, the algorithm selects $x_3^g =$ `/a` as none of the more specific instructions (`/a[id="t1"]` or `/a[id="t2"]`) has a support greater than $\delta = 1/2$. The resulting unified XPath is $x =$ `/div[class="title"]/span/a`.

Algorithm 2: Top-Down XPath Unification

Input: set X of sequential XPaths
Input: support function *support* : $X \to \mathbb{N}$
Input: threshold δ
TS = *support*(X)
$k = \max_{x \in X} |x|$
$X^0 = X$
foreach $i = 1, \ldots, k$ **do**
  $I^i = \{x_i \mid x \in X^{i-1}\}$
  if $I^i = \emptyset$ **then**
    $i = i - 1$
    break
  foreach $B \subseteq F$ *in decreasing order of* $|B|$ **do**
    $support_B =$ FindSupport$(x_B, X^{i-1}, i,$ *support*$)$
    if $support_B > \delta \cdot$ *TS* **then**
      $x_i^g = x_B$
      $X^i = \{x \in X^{i-1} \mid x_i \text{ agrees with } x_B\}$
      break
return $x_1^g, \ldots, x_i^g$

6 Crawler Synthesis

In this section we complete the description of our crawler synthesizer. To do so, we describe the synthesis of a page crawler for each website s. Recall that a page crawler corresponds to a URL pattern $U(s)$ which defines the webpages of interest. The synthesis of a page crawler is intertwined with the data extractor synthesis, and uses similar unification techniques to generate the URL pattern.

Initialization We assume that each website $s \in S$ is given by a "main" webpage $p_{main}(s)$. Initially, the set $P(s)$ of webpages of s is the set of all webpages obtained by following links in $p_{main}(s)$ and recursively following links in the resulting pages, where the traversed links are selected based on some heuristic function which determines which links are more likely to lead to relevant pages.

Iterations We apply the data extractor synthesis algorithm of Section 5 using the sets $P(s)$. At the end of phase 2 of each iteration, we update $U(s)$ using the steps described below. At the beginning of phase 1 of the subsequent iteration we then update $P(s)$ to the set of webpages whose URLs conform with $U(s)$.

(6) *Filtering webpage sets:* Based on the observation that relevant webpages of a website s have a similar structure, we keep in $P(s)$ only webpages that contain container and attribute nodes that match the generated $E(s)$ and are reachable from $p_{main}(s)$ via such webpages.

(7) *Generating URL patterns:* For each webpage $p \in P(s)$ we consider its URL. We unify the URLs into $U(s)$ by a variation of Algorithm 2 which views a URL as a sequence of instructions, similarly to a sequential XPath.

7 Evaluation

In this section we evaluate the effectiveness of our approach. We used it to synthesize data extracting web-crawlers for real-world websites containing structured data of different categories. Our experiments focus on two different aspects: (i) the ability to successfully synthesize web-crawlers, and (ii) the performance of the resulting web crawlers.

7.1 Experimental Settings

We have implemented our tool in C#. All experiments ran on a machine with a quad core CPU and 32GB memory. Our experiments were run on 30 different websites, related to nine different categories: books, TVs, conferences, universities, cameras, phones, movies, songs and hotels. For each category we selected a group of 3-4 known sites, which appear in the first page of Google search results.

The sites in each category have a different structure, but they share at least some of their instances, which makes our approach applicable. The complexity of the data extracted from different categories is also different. For instance a movie has four attributes: *title*, *genre*, *director* and list of *actors*. For a book, the set of attributes consists of *title*, *author* and *price*, while the attribute set of a camera consists of the *name* and *price* only. In each category we used one manually written crawler and automatically synthesized the others (for the books category we also experimented with 3 partial extraction schemes, one for each attribute). To synthesize the web crawlers, our tool processed over $12,000$ webpages from the 30 different sites.

To evaluate the effectiveness of our tool we consider 4 aspects of synthesized crawlers: (i) Crawling scheme completeness, (ii) URL filtering, (iii) Container extraction, and (iv) Attributes extraction.

7.2 Experiments and Results

Crawling Scheme Completeness A complete crawling scheme defines extraction queries for all of the data attributes. The completeness of the synthesized crawling schemes is an indicator for the success of our approach in synthesizing crawlers. To measure completeness, we calculated for each category the average number of attributes covered by the schemes, divided by the number of attributes of the category. The results are reported in Fig. 6 (left). The results show that the resulting extraction schemes are mostly complete, with a few missing attribute extraction queries.

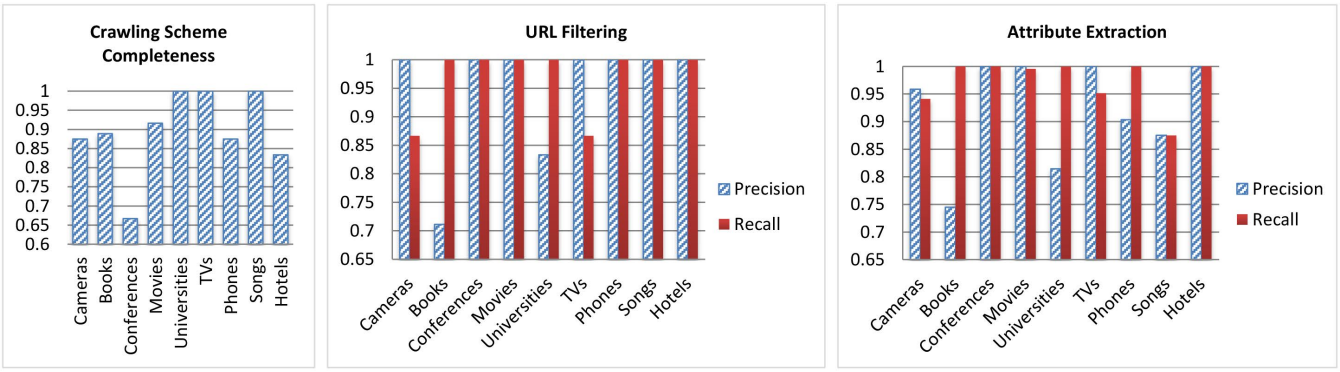

Figure 6: Results: Crawling scheme completeness (left), URL filtering (middle) and Attribute extraction (right) for each category.

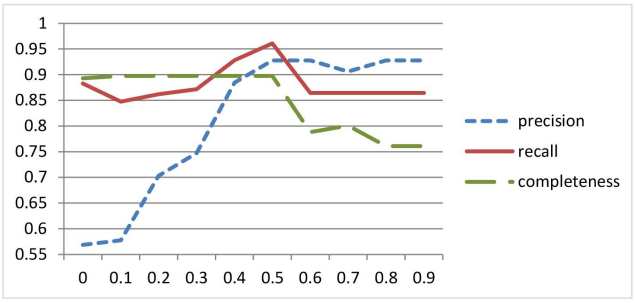

Figure 7: Attribute extraction precision and recall, and crawling scheme completeness, as a function of the threshold of Jaccard similarity used to define equivalence between instances.

URL Filtering The ability to locate pages containing data is an important aspect of a crawler's performance. To evaluate the URL filtering performance of the synthesized crawlers, we measure the *recall* and *precision* of the synthesized URL pattern for each site:

$$recall = \frac{|Rel \cap Sol|}{|Rel|} \quad precision = \frac{|Rel \cap Sol|}{|Sol|} \qquad (1)$$

To do so, we have manually generated two sets of URLs for each site: one containing URLs for pages that contain relevant data, comprising the Rel set (ground truth), and another, denoted Irr, contains a mixture of irrelevant URLs from the same site. Sol contains the URLs from $Rel \cup Irr$ that match the synthesized URL pattern for the site (i.e., the URLs accepted by the synthesised URL pattern). A good performing URL filtering pattern should match all the URLs from Rel and should not match any from Irr. The average recall and precision scores of the sites of each category are calculated and reported in Fig. 6 (middle).

Container Extraction To check the correctness of the synthesized container extraction query, we have manually reviewed the resulting container XPaths against the HTML sources of the relevant webpages for each site, to verify that each extracted container contains exactly one data item. We found that the containers always contained no more than one item. However, in a few cameras and songs websites, the container query was too specific and did not extract some of the containers (this happened in tables containing class="odd" in some rows and class="even" in others), which affected the recall scores of attribute extraction.

Attributes Extraction We calculate the recall and precision (see equation (1)) of the extraction query for each attribute. Technically, for each category of sites, we have manually written extraction queries for each attribute in every one of the category related sites. For each attribute a, we used these extraction queries to extract the instances of a from a set of sample pages from each site. The extracted instances are collected in Rel. We have also applied the synthesized extraction queries (as a concatenation of the container XPath and attribute XPath) to extract instances of a from the same pages into Sol. For each site, the precision and recall are calculated according to equation (1). The average (over sites of the same category) recall and precision scores of all attributes of each category are reported in Fig. 6 (right).

Equivalence Relation To evaluate the effect of the threshold used in the equivalence relation, $\equiv_a$, on the synthesized crawlers, we have measured the average completeness, as well as the average recall and precision scores of attribute extraction as a function of the threshold. The results appear in Fig. 7.

Remark. The reported attribute extraction recalls in Fig. 6 and Fig. 7 are computed based on queries for which synthesis succeeded (missing queries affect only completeness, and not recall).

7.3 Discussion

The completeness of the synthesized extraction schemes is highly dependent upon the ability to identify instances in pages of some site by comparison to instances gathered from other sites. For most categories, completeness is high. For the conferences category, however, completeness is low. This is due to the use of acronyms in conference names (e.g., ICSE) in some sites vs. full names (e.g., International Conference on Software Engineering) in others, which makes it hard for our syntax-based equivalence relation to identify matches. This could be improved by using semantic equivalence relations (such as ESA [12] or W2V [33]).

As for the quality of the resulting extraction schemes and URL filtering patterns, most of the categories have perfect recall (Fig. 6). However, some have a slightly lower recall due to our attempt to keep the synthesized XPaths (or regular expressions, for URL filtering) as concrete as possible while having majority agreement. This design choice makes our method tolerant to small noises in the identified data instances, and prevents such noises from causing drifting, without negative examples. Yet, in some cases, the resulting XPaths are too specific and result in a sub-optimal recall.

For precision, most categories have good scores, while a few have lower scores. Loss of precision can be attributed to the majority-based unification and the lack of negative examples. For the books category, for instance, the synthesized extraction XPath of price for some sites is too general, since they list multiple price instances (original price, discount amount, and new price). All are listed in

the same "parent container" with the author and book title, and are therefore not filtered by the container, hence affecting XPath unification. This could be improved with user guidance.

The results in Fig. 7 reflect the tradeoff between precision and crawling scheme completeness. A more strict equivalence relation (with higher threshold) leads to a better precision but has negative effect on the scheme completeness, whereas the use of a forgiving equivalence relation (with lower threshold) severely affects the precision. We use a threshold of 0.5 as a balanced threshold value. According to our findings, the attribute queries suffer from a low recall for both low and high threshold values. In low threshold, it is due to wrong queries, that extract wrong nodes (e.g., menu nodes), without including attribute nodes. For higher threshold values, the tool identified less instances of attribute nodes (sometimes only one), leading to a lower quality generalization.

Real-World Use Case We used our crawler synthesis process as a basis for data extraction for several product reviews websites. For instance, tvexp.com, weppir.com, camexp.com and phonesum.com extract product names and specifications (specs) using our approach. We manually added another layer of specs scoring, and created comparison sites for product specs. These websites have a continually updated database with over 20,000 products.

8 Related Work

In this section, we briefly survey closely-related work. While there has been a lot of past work on various aspects of mining and data extraction, our technique has the following unique combination of features: (i) works across multiple websites, (ii) synthesizes both the extraction XPath queries, and the URL pattern, (iii) is automatic and does not require user interaction, (iv) works with only positive examples, (v) does not require an external database, and (vi) synthesizes a working crawler.

Data Mining and Wrapper Induction Our work is related to data mining and wrapper induction. In contrast to supervised techniques (e.g., [24, 25, 29, 5, 16]), our approach only requires an initial crawler (or partial crawling scheme) and requires no tagged examples. FlashExtract [25] allows end-users to give examples via an interaction model to extract various fields and to link them using special constructs. It then applies an inductive synthesis algorithm to synthesize the intended data extraction program from the given examples. In contrast, our starting point is a crawler for one (or more) sites, which we then extrapolate from. Further, our technique only requires positive examples (obtained by bootstrapping our knowledge base by crawling other sites).

Unsupervised extraction techniques [2, 3, 8, 32, 31, 34, 40, 36] have been proposed. Several works [3, 31, 2, 10, 38, 27, 37] propose methods that use repeated pattern mining to discover data records, while [34, 40] use tree-edit distance as the basis for record recognition and extraction in a single given page. These methods require manual annotation of the extracted data or rely on knowledge bases [14, 20]. Roadrunner [8] uses similarities and differences between webpages to discover data extraction pattern. Similarities are used to cluster similar pages together and dissimilarities between pages in the same cluster are used to identify relevant structures. Other information extraction techniques rely on textual, or use visual features of the document [41, 30] for data extraction. ClustVX [15] renders the webpage in contemporary web browser, for processing all visual styling information. Visual and structural features are then used as similarity metric to cluster webpage elements. Tag paths of the clustered webpages are then used to derive extraction rules. In contrast, our approach does not use visual styling, but relies on similar content between the different sites.

HAO et al. [18] present a method for data extraction from a group of sites. Their method is based on a classifier that is trained on a seed site using a set of predefined feature types. The classifier is then used as a base for identification and extraction of attribute instances in unseen sites. In contrast, our goal is to synthesize XPaths that are human-readable, editable, and efficient. Further, with the lack of an attribute grouping mechanism (such as our notion of container), the method cannot handle pages with multiple data items.

Program Synthesis Several works on automatic synthesis of programs [22, 13, 25, 17] were recently proposed, aiming for automating repetitive programming tasks. Programming by example, for instance, is a technique used in [22, 25] for synthesizing a program by asking the user to demonstrate actions on concrete examples. Inspired by these works, our approach automatically synthesizes data extracting web crawlers. However, we require no user interaction.

Semantic Annotation Many works in this area attempt to automatically annotate webpages with semantic meta-data.

Seeker [11] is a platform for large-scale text analysis, and an application written on the platform called SemTag that performs automated semantic tagging of large corpora. Ciravegna et al. [6] propose a methodology based on adaptive information extraction and implement it in a tool called Armadillo [4]. The learning process is seeded by a user defined lexicon or an external data source. In contrast to these works, our approach does not require external knowledge base and works by bootstrapping its knowledge base.

Other Aspects of Web Crawling There are a lot of works dealing with different aspects of web crawlers. Jiang et al. [23] and Jung et al. [1] deal with deep-web related issues, like the problem of discovering webpages that cannot be reached by traditional web crawlers mostly because they are results of a query submitted to a dynamic form and they are not reachable via direct links from other pages. Some other works like [35, 39] address the problem of efficient navigation of website pages to reach pages of specific type by training a decision model and using it do decide which links to follow in each step. Our paper focuses on the different problem of data extraction, and is complementary to these techniques.

9 Conclusion

We presented an automatic synthesis of data extracting web crawlers by extrapolating existing crawlers for the same category of data from other websites. Our technique relies only on data overlaps between the websites and not on their concrete representation. As such we manage to handle significantly different websites. Technically, we automatically label data in one site based on others and synthesize a crawler from the labeled data. Unlike techniques that synthesize crawlers from user provided annotated data, we cannot assume that all annotations are correct (hence some of the examples might be false positives), and we cannot assume that unannotated data is noise (hence we have no negative examples). We overcome these difficulties by a notion of containers that filters the labeling.

We have implemented our approach and used it to automatically synthesize 30 crawlers for websites in nine different product categories. We used the synthesized crawlers to crawl more than $12,000$ webpages over all categories. In addition, we used our method to build crawlers for real product reviews websites.

Acknowledgements

The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement no. 615688, ERC-COG-PRIME.

10 References

[1] AN, Y. J., GELLER, J., WU, Y.-T., AND CHUN, S. Semantic deep web: automatic attribute extraction from the deep web data sources. In *Proceedings of the 2007 ACM symposium on Applied computing* (2007), ACM, pp. 1667–1672.

[2] ARASU, A., AND GARCIA-MOLINA, H. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), ACM, pp. 337–348.

[3] CHANG, C.-H., AND LUI, S.-C. IEPAD: Information extraction based on pattern discovery. In *Proceedings of the 10th International Conference on World Wide Web* (2001), WWW '01, pp. 681–688.

[4] CHAPMAN, S., DINGLI, A., AND CIRAVEGNA, F. Armadillo: harvesting information for the semantic web. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval* (2004), ACM, pp. 598–598.

[5] CHUANG, S.-L., AND HSU, J.-J. Tree-structured template generation for web pages. In *Web Intelligence, 2004. WI 2004. Proceedings. IEEE/WIC/ACM International Conference on* (2004), IEEE, pp. 327–333.

[6] CIRAVEGNA, F., CHAPMAN, S., DINGLI, A., AND WILKS, Y. Learning to harvest information for the semantic web. In *The Semantic Web: Research and Applications*. Springer, 2004, pp. 312–326.

[7] CLARK, J., DEROSE, S., ET AL. Xml path language (xpath). *W3C recommendation 16* (1999).

[8] CRESCENZI, V., MECCA, G., AND MERIALDO, P. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases* (2001), VLDB '01, pp. 109–118.

[9] DALVI, N., BOHANNON, P., AND 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* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 335–348.

[10] DALVI, N., KUMAR, R., AND SOLIMAN, M. Automatic wrappers for large scale web extraction. *Proceedings of the VLDB Endowment 4*, 4 (2011), 219–230.

[11] DILL, S., EIRON, N., GIBSON, D., GRUHL, D., GUHA, R., JHINGRAN, A., KANUNGO, T., RAJAGOPALAN, S., TOMKINS, A., TOMLIN, J. A., ET AL. Semtag and seeker: Bootstrapping the semantic web via automated semantic annotation. In *Proceedings of the 12th international conference on World Wide Web* (2003), ACM, pp. 178–186.

[12] GABRILOVICH, E., AND MARKOVITCH, S. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *IJCAI* (2007), vol. 7, pp. 1606–1611.

[13] GALENSON, J., REAMES, P., BODIK, R., HARTMANN, B., AND SEN, K. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 653–663.

[14] GENTILE, A. L., ZHANG, Z., AUGENSTEIN, I., AND CIRAVEGNA, F. Unsupervised wrapper induction using linked data. In *Proceedings of the Seventh International Conference on Knowledge Capture* (New York, NY, USA, 2013), K-CAP '13, ACM, pp. 41–48.

[15] GRIGALIS, T. Towards web-scale structured web data extraction. In *Proceedings of the sixth ACM international conference on Web search and data mining* (2013), ACM, pp. 753–758.

[16] GULHANE, P., MADAAN, A., MEHTA, R., RAMAMIRTHAM, J., RASTOGI, R., SATPAL, S., SENGAMEDU, S. H., TENGLI, A., AND TIWARI, C. Web-scale information extraction with vertex. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (2011), IEEE, pp. 1209–1220.

[17] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Synthesis of loop-free programs. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 62–73.

[18] HAO, Q., CAI, R., PANG, Y., AND 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* (2011), ACM, pp. 775–784.

[19] HAWKINS, P., AIKEN, A., FISHER, K., RINARD, M. C., AND SAGIV, M. Data structure fusion. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010* (2010), pp. 204–221.

[20] HONG, J. L. Data extraction for deep web using wordnet. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 41*, 6 (2011), 854–868.

[21] JACCARD, P. The distribution of the flora in the alpine zone. *New Phytologist 11*, 37–50.

[22] JHA, S., GULWANI, S., SESHIA, S., TIWARI, A., ET AL. Oracle-guided component-based program synthesis. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (2010), vol. 1, IEEE, pp. 215–224.

[23] JIANG, L., WU, Z., FENG, Q., LIU, J., AND ZHENG, Q. Efficient deep web crawling using reinforcement learning. In *Advances in Knowledge Discovery and Data Mining*. Springer, 2010, pp. 428–439.

[24] KUSHMERICK, N., WELD, D. S., AND DOORENBOS, R. B. Wrapper induction for information extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes* (1997), pp. 729–737.

[25] LE, V., AND GULWANI, S. Flashextract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 55.

[26] LEOTTA, M., STOCCO, A., RICCA, F., AND TONELLA, P. Reducing web test cases aging by means of robust xpath locators. In *Proceedings of 25th International Symposium on Software Reliability Engineering Workshops (ISSREW 2014)* (2014), pp. 449–454.

[27] LIU, B., GROSSMAN, R., AND ZHAI, Y. Mining data records in web pages. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (2003), ACM, pp. 601–606.

[28] LIU, D., WANG, X., YAN, Z., AND LI, Q. Robust web data extraction: a novel approach based on minimum cost script edit model. In *Web Information Systems and Mining*. Springer, 2012, pp. 497–509.

[29] LIU, L., PU, C., AND HAN, W. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Data Engineering, 2000. Proceedings. 16th International Conference on* (2000), IEEE, pp. 611–621.

[30] LIU, W., MENG, X., AND MENG, W. Vide: A vision-based approach for deep web data extraction. *Knowledge and Data Engineering, IEEE Transactions on 22*, 3 (2010), 447–460.

[31] MIAO, G., TATEMURA, J., HSIUNG, W.-P., SAWIRES, A., AND MOSER, L. E. Extracting data records from the web using tag path clustering. In *Proceedings of the 18th International Conference on World Wide Web* (New York, NY, USA, 2009), WWW '09, ACM, pp. 981–990.

[32] MICHELSON, M., AND KNOBLOCK, C. A. Unsupervised information extraction from unstructured, ungrammatical data sources on the world wide web. *International Journal of Document Analysis and Recognition (IJDAR) 10*, 3-4 (2007), 211–226.

[33] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[34] REIS, D. D. C., GOLGHER, P. B., SILVA, A. S., AND LAENDER, A. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 502–511.

[35] RENNIE, J., MCCALLUM, A., ET AL. Using reinforcement learning to spider the web efficiently. In *ICML* (1999), vol. 99, pp. 335–343.

[36] SLEIMAN, H. A., AND CORCHUELO, R. Tex: An efficient and effective unsupervised web information extractor. *Knowledge-Based Systems 39* (2013), 109–123.

[37] THAMVISET, W., AND WONGTHANAVASU, S. Information extraction for deep web using repetitive subject pattern. *World Wide Web* (2013), 1–31.

[38] THAMVISET, W., AND WONGTHANAVASU, S. Information extraction for deep web using repetitive subject pattern. *World Wide Web 17*, 5 (2014), 1109–1139.

[39] VYDISWARAN, V. V., AND SARAWAGI, S. Learning to extract information from large websites using sequential models. In *COMAD* (2005), pp. 3–14.

[40] ZHAI, Y., AND LIU, B. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web* (2005), ACM, pp. 76–85.

[41] ZHU, J., NIE, Z., WEN, J.-R., ZHANG, B., AND 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* (New York, NY, USA, 2006), KDD '06, ACM, pp. 494–503.