Interactive Wrapper Generation with Minimal User Effort*

Utku Irmak
CIS Department
Polytechnic University
Brooklyn, NY 11201
uirmak@cis.poly.edu

Torsten Suel
CIS Department
Polytechnic University
Brooklyn, NY 11201
suel@poly.edu

ABSTRACT

While much of the data on the web is unstructured in nature, there is also a significant amount of embedded structured data, such as product information on e-commerce sites or stock data on financial sites. A large amount of research has focused on the problem of generating wrappers, i.e., software tools that allow easy and robust extraction of structured data from text and HTML sources. In many applications, such as comparison shopping, data has to be extracted from many different sources, making manual coding of a wrapper for each source impractical. On the other hand, fully automatic approaches are often not reliable enough, resulting in low quality of the extracted data.

We describe a complete system for semi-automatic wrapper generation that can be trained on different data sources in a simple interactive manner. Our goal is to minimize the amount of user effort for training reliable wrappers through design of a suitable training interface that is implemented based on a powerful underlying extraction language and a set of training and ranking algorithms. Our experiments show that our system achieves reliable extraction with a very small amount of user effort.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services

General Terms

Algorithms, Design, Experimentation.

Keywords

Data extraction, active learning, wrapper generation.

1. INTRODUCTION

Over the last decade, the world-wide web (WWW) has become one of the most widely used information resources. On the WWW, the information is usually presented via Hypertext Markup Language (HTML) to make its perception easier for humans. However, this visual presentation is often not appropriate for automatic processing. Thus, it is often necessary to extract the data embedded in the pages into a relational or other structured format for further processing.

This task is usually performed by software tools called *wrappers*. The goal of a wrapper is to translate the relevant data embedded in web pages into a relational (or other regular) structure. Wrappers may be constructed manually, or by a semi-automatic or automatic approach. Since the manual generation of wrappers is tedious and error-prone, semi-automatic and automatic wrapper construction systems are highly preferable; see, e.g., [4, 5, 16, 22]. Common applications of wrappers include comparison shopping where information from multiple online shops is extracted to compare prices, and online monitoring of news over multiple websites or bulletin boards, say for items relevant to a particular topic.

In this paper, we describe a new system for semi-automatic wrapper generation. The system was originally developed as a prototype for an industrial application that required data extraction from multiple public websites that frequently modify the format of their pages, and a manual approach to this problem required a significant ongoing effort. (We are unable to disclose more details of the application scenario.) The system provides a visual interface and interactively generates a wrapper from input provided by a human operator (user). Unlike in some earlier systems, the user is not required to work on HTML source code, Document Object Model (DOM) trees, or some intermediary expression generated by the system. All interaction takes place in the browser on the original view of the web pages, by using the mouse. The constructed wrappers are internally represented in a powerful special-purpose declarative extraction language, but the user does not need to read or understand this language.

Our system uses various algorithmic techniques hidden behind the user interface in order to minimize the amount of manual input provided by the user. Often only one example is required to generate a correct wrapper. Since labeling of training examples is considered the major bottleneck in wrapper generators, this approach offers great advantages. In addition, hiding all technical details behind the interface makes the resulting tool very easy to learn and use.

As part of our approach, we present an active-learning based framework that utilizes a larger set of unlabeled web pages during wrapper generation. Within this framework, we perform ranking of different hypotheses about the extraction scenario using a category utility function [10]. This allows us to significantly increase the accuracy and robustness of the generated wrappers through a few simple user interactions that require much less effort than manual labeling of additional examples. The approach offers an elegant solution to common obstacles to the generation of robust wrappers, such as missing (or multiple) attribute values or different attribute orderings in a tuple as described in [11], as well as other representational variations between different tuples, say, when a site uses

*Work supported by NSF Awards IDM-0205647 and CCR-0093400, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

fonts, colors, or indentation to convey certain information (e.g., by showing a stock price in red if there was a decrease). Such variations often lead to problems later if a particular case was not encountered among the few labeled examples that are provided. In our approach, these variations are usually resolved in a fairly painless manner during wrapper generation, and the risk of later wrapper failures is reduced significantly. Combined with the interactive interface and powerful internal extraction language, this allows us to deal with challenging extraction scenarios that are often impossible to capture under previous approaches.

1.1 Problem Statement and Motivation

We now formally introduce the problem we consider and discuss it on a simple example. In the wrapper generation problem, our goal is to correctly extract a set of tuples from a class of web pages available from one or several web sites. For example, a meta search tool might have to extract tuples from result pages generated by the Yahoo! search engine; see Figure 1.1 for the query "parallel databases". In particular, we might want to extract the following four attributes for each of the top-10 results: (i) the rank of the page, (ii) the page title, (iii) the page URL, and (iv) the *snippet* of surrounding text provided by the engine. Thus, in the example the first tuple would be *(1, "Parallel and Distributed Databases", www.csse.monash.edu.au/~rama/cse3000/notes/pdb.pdf, "... Introduction. Parallel and Distributed databases are. being increasingly used to achieve, ... Parallel Database Architecture. Parallel databases (Pdbs) use more ...").* (We choose the Yahoo! example for its simplicity. Some search engines now provide APIs that provide result data in XML format, but this option is not available on many other sites with embedded data.)

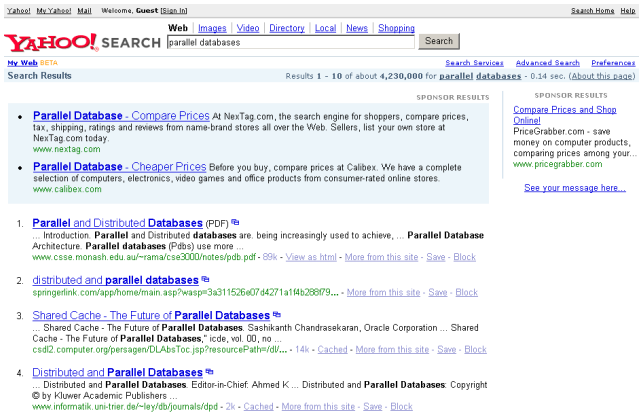


Figure 1.1: A result page from the Yahoo! search engine

In the standard semi-automatic approach to this problem, the user first defines the structure of the resulting tuples (i.e., the names and maybe the types of the attributes), and then identifies a few *training tuples* on one or several example pages, say by highlighting the attribute values of these tuples directly on the page using the mouse. Thus, the user supplies a small training set, and the system then tries to induce a wrapper (set of extraction rules) for extracting tuples from these pages as well as other pages of the same class that may be encountered in the future (i.e., Yahoo! result pages for future queries). Depending on the scenario, there may be a single tuple or a large number of tuples on each page, and in the latter case the user should not have to label every tuple on the page. Most approaches only require positive examples (tuples), and the system will then make sure not to consider any overly general wrappers that extract everything on the page. In the Yahoo! example, after highlighting the first tuple, the system might consider the sponsored links at the top or the ads on the right as *possible tuples*, but probably not the text above the search box or other very dissimilar structures.

Some remarks now on the resulting challenges. First, the extraction rules themselves are expressed in terms of some underlying language that needs to be powerful enough to capture the scenario. For example, a simple choice would be to define the start of each attribute that needs to be extracted by evaluating a regular expression on the HTML of the Yahoo! result page, but depending on the scenario more powerful languages may be needed that take the DOM tree structure of the HTML or even the layout of the rendered page into account.

Second, as we see from the example, the structure of the result tuples shows several variations: The second result has an empty snippet text, while the first result has an extra label "PDF" at the end of the page title to indicate PDF format. We also have sponsored results and ads that are similar to real results in their structure but that we may not wish to extract. Other variations not shown in Figure 1.1 occur when a search term is a stock symbol or matches a current popular news items, or when two consecutive results are from the same site. Thus, if only a small training set of tuples is provided, it is likely that the generated extraction rules will fail on future pages that contain variations not occurring in the training set. On the other hand, a large training set involves a significant amount of user effort.

Our goal is to overcome these problems by using what is essentially an active learning approach, where the system is supplied with a very small set of initial training examples (usually just a single tuple) plus an unlabeled larger set of other result pages (hopefully containing most of the common variations) called the *verification set*. Such a set can usually be obtained by crawling parts of the targeted site or by automatically issuing a few queries. After analyzing this data, the system then interactively requests a very small amount of additional input from the user as needed to resolve any uncertain cases in the verification set. We will show that such an approach, when combined with a powerful underlying extraction languages that takes DOM tree structure as well as the layout of the rendered result page into account, results in robust extraction with only minimal user effort.

The remainder of this paper is organized as follows. In the next section, we describe our basic approach and its operation on an example. Section 3 summarizes and discusses our contributions. Sections 4 to 6 describe the design of the system and the underlying algorithmic techniques, and Section 7 contains our experimental results. Finally, Section 8 gives an overview of related work, and Section 9 provides some concluding remarks. Due to space constraints, many details are omitted from this manuscript; see [12] for the full version.

2. INTERACTIVE WRAPPER GENERATION

We now discuss the steps involved in generating a wrapper. In later sections we will show how to implement these steps at the lower level using various algorithmic and learning-based techniques.

2.1 The Interactive Interface

Our system provides a visual interface within Internet Explorer that enables the user to work on the original view of the web pages. The user starts the process by supplying the number and names of attributes in a tuple, and optionally also their types. (Providing types, and possibly also constraints such as "not empty" or "$\leq$ 100", can help in some challenging scenarios, but is usually not needed. All our experiments later are run without providing types.) The user then inputs a *training tuple* on a page by highlighting each

attribute with the mouse. The user can highlight any one of the complete tuples (not necessarily the first one) from the desired tuple set she wishes to extract. The desired tuple set depends on the needs of the application, and reflects a specific *extraction scenario*.

Since highlighting tuples by hand is tedious, our goal is to minimize this effort, and our system usually only requires highlighting of a single training tuple on a page containing many tuples. Once a tuple is marked, the system attempts to *predict* the desired tuple set on the page, by determining the set of all possible tuples (i.e., all structures in the page that are "similar enough" to the highlighted tuple) and then considering various subsets of the possible tuples that each contain the highlighted tuple. Each subset that is considered is called a *candidate tuple set*, and the goal is to have the user select which candidate tuple set is the desired tuple set.

Thus, the system needs to generate a collection of candidate tuple sets that is likely to contain the desired tuple set but not so large that we cannot find the desired set among all the other sets. The key is to exploit the structure of the set of possible tuples, such that ideally each candidate tuple set corresponds to a reasonable extraction scenario on the page, though maybe not the one the user has in mind. (E.g., extracting both the top-10 results and sponsored results on the page in Figure 1.1 might be a reasonable goal in some other scenarios.) Informally, a wrapper can be seen as an extraction pattern, for example a regular expression on the HTML source or some expression on the DOM tree or visual structure of the page. Given a training tuple, there are many extraction patterns that correctly extract the training tuple – plus possibly other tuples on the page. By considering multiple such patterns, we generate multiple candidate tuple sets corresponding to different extraction scenarios.

Thus, each candidate tuple set that is considered is the result of applying a wrapper to the training web page. After the user highlights a single training tuple on the page, our system generates many different extraction patterns that may identify a number of different candidate tuple sets on the training web page. The precise internal structure of our wrappers is actually quite complex, see Section 5.4, but the important thing for now is that a wrapper consists of a set of extraction patterns that agree with each other on the training page, i.e., extract exactly the same candidate tuple set on this page. Each pattern essentially corresponds to a different hypothesis [23] about what makes something a tuple. Thus, a wrapper groups and stores all those patterns that agree on the current page. Some of these patterns may be too specific, with odd conditions that just happen to be true on the current page, while others may be overly general.

Our system generates and renders several versions of the training web page, where each version highlights a different candidate tuple set extracted by a different wrapper. The user can now navigate between these pages using special toolbar buttons added to the browser, and choose the desired tuple set. If none of the pages shows the desired set, the user may correct this problem by either highlighting another tuple or editing one of the displayed sets. We add one additional important idea to allow the user to efficiently select the correct tuple set: Navigating all choices would be quite inefficient if there are many different candidate sets. We address this by ranking the candidate tuple sets using a technique described later, such that more likely sets are ranked higher. As we show in our experiments, in most cases the correct set is displayed among the very first sets.

Once the user selects the desired tuple set, the system treats this as a confirmation for the corresponding wrapper. Note that while all extraction patterns in the wrapper are consistent on the current training page, they might diverge on unseen web pages (in which case some of the extraction patterns will turn out to be incorrect). Possible reasons are variations in the web pages, such as missing attributes and variant attribute permutations in some tuples, that may not occur in any page considered thusfar. This leads us to the idea of using an additional *verification set* of unlabeled web pages from the same site to increase the accuracy and robustness of the generated wrapper. All extraction patterns in the wrapper are tested on the verification set to see if any of them disagree on any page (i.e., extract a different set of tuples). If so, the system interacts with the user by displaying, again in suitably ranked order, the different tuple sets on the page with the most disagreements, and asking the user for a choice. After the user selects the desired set, the non-conforming patterns are removed from the wrapper and the new wrapper is again tested until there are no more disagreements. The final wrapper is then stored for later use on not yet seen pages.

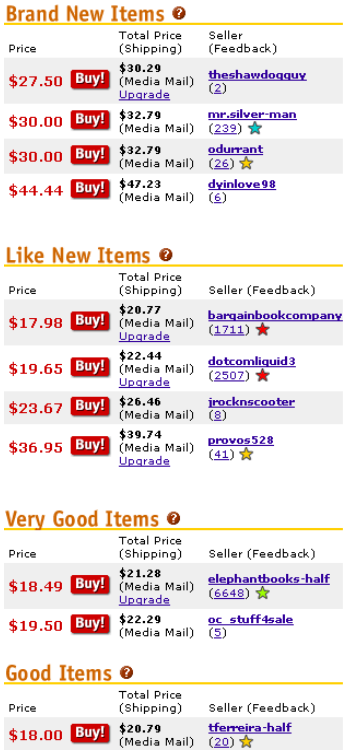
As we will show, the described framework significantly increases the accuracy and robustness of the resulting wrapper. The total extra user effort is small, typically just a few mouse clicks, and is needed only if disagreements are found in the verification set. The verification set can usually be acquired in an almost automatic way, by pointing a crawler to parts of the website likely to contain tuples (even if only some of the pages contain tuples while others are unrelated). If a user decides not to provide a verification set, or to provide a set that is too small, it becomes much more likely that disagreements will be encountered later during the operation of the system. In the case of such later disagreements, the system can be set up to take any of the following actions depending on the scenario: (i) stop immediately, (ii) ignore the current page, (iii) choose the most likely tuple set using built-in heuristics and continue to extract tuples with a modified wrapper, or (iv) store all different tuple sets separately for future analysis. In addition, the system notifies an administrator.

In summary, our method typically requires the labeling of a single tuple, followed by a selection of a tuple set from a ranked list where the desired set is usually among the first few, plus the labeling of another tuple in the rare case when the desired set is not found in the list. If a verification set is provided, then any disagreement of the extraction patterns in the current wrapper on a page in this set results in another selection from a usually much shorter ranked list.

2.2 An Example

We now illustrate this process using a real web page from the site `www.half.ebay.com`, shown in Figure 2.1. This page provides an example of a fairly challenging extraction task on which many existing wrapper generation systems would likely fail. The user might be interested in collecting postings of auction data from this website over a period of time, e.g., to perform statistical analyses on prices, sellers, or shipping methods. Let us assume the user is interested in extracting tuples with the attributes "Price", "Total-Price", and "Seller" for certain products. In general, there are multiple different reasonable extraction scenarios on these web pages, corresponding to different user needs and resulting in different tuple sets. For example, one scenario is to extract all possible such tuples on the web page, or only those tuples that appear in the table "Like New Items", or the first tuples of each table (i.e., only the cheapest items since the entries are ordered by price). Although it may not look reasonable for this web page, another scenario might be to extract all tuples whose background color is gray and not white. This might be appropriate in cases where this color conveys some distinct meaning, as is in fact the case on many other sites.

Let us assume a user is interested in only those tuples that appear under "Like New Items" and where the seller has a "Red Star" in "Feedback Rating". (We point out that in Figure 2.1 only the 3rd and 4th stars from the top are red.) On the website, stars with various colors represent the feedback profiles of the sellers, and "Red



Brand New Items

Price		Total Price (Shipping)	Seller (Feedback)
$27.50	Buy!	$30.29 (Media Mail) Upgrade	theshawdoqguy (2)
$30.00	Buy!	$32.79 (Media Mail)	mr.silver-man (239)
$30.00	Buy!	$32.79 (Media Mail)	odurrant (26)
$44.44	Buy!	$47.23 (Media Mail)	dyinlove98 (6)

Like New Items

Price		Total Price (Shipping)	Seller (Feedback)
$17.98	Buy!	$20.77 (Media Mail) Upgrade	bargainbookcompany (1711)
$19.65	Buy!	$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507)
$23.67	Buy!	$26.46 (Media Mail)	jrocknscooter (8)
$36.95	Buy!	$39.74 (Media Mail) Upgrade	provos528 (41)

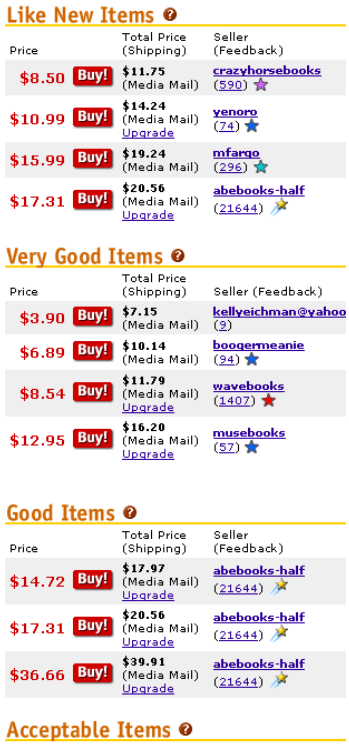
Very Good Items

Price		Total Price (Shipping)	Seller (Feedback)
$18.49	Buy!	$21.28 (Media Mail) Upgrade	elephantbooks-half (6648)
$19.50	Buy!	$22.29 (Media Mail)	oc_stuff4sale (5)

Good Items

Price		Total Price (Shipping)	Seller (Feedback)
$18.00	Buy!	$20.79 (Media Mail)	tferreira-half (20)

Figure 2.1: Training page from `www.half.ebay.com`.



Like New Items

Price		Total Price (Shipping)	Seller (Feedback)
$8.50	Buy!	$11.75 (Media Mail)	crazyhorsebooks (590)
$10.99	Buy!	$14.24 (Media Mail) Upgrade	venoro (74)
$15.99	Buy!	$19.24 (Media Mail)	mfargo (296)
$17.31	Buy!	$20.56 (Media Mail) Upgrade	abebooks-half (21644)

Very Good Items

Price		Total Price (Shipping)	Seller (Feedback)
$3.90	Buy!	$7.15 (Media Mail)	kellyeichman@yahoo (9)
$6.89	Buy!	$10.14 (Media Mail)	boogermeanie (94)
$8.54	Buy!	$11.79 (Media Mail) Upgrade	wavebooks (1407)
$12.95	Buy!	$16.20 (Media Mail) Upgrade	musebooks (57)

Good Items

Price		Total Price (Shipping)	Seller (Feedback)
$14.72	Buy!	$17.97 (Media Mail) Upgrade	abebooks-half (21644)
$17.31	Buy!	$20.56 (Media Mail) Upgrade	abebooks-half (21644)
$36.66	Buy!	$39.91 (Media Mail) Upgrade	abebooks-half (21644)

Acceptable Items

Figure 2.2: Page where 2nd disagreement occurs.

Star" represents a "Feedback Profile of 1,000 to 4,999 users".

We started the training process on the page shown in Figure 2.1, using a verification set of size ten. When we highlighted one complete tuple on this training web page according to the desired extraction scenario (the first tuple under "Like New Items"), the system identified 18 different candidate tuple sets. These sets were ranked from most to least likely according to techniques described later, and the desired set was ranked as the third (behind options of *all tuples* and *all "Like New Items"*). After selecting the desired set, the system tested the wrapper on the verification set, resulting in two disagreements.

The first disagreement was on a page where the wrapper resulted in two different tuple sets. The reason for this is interesting: We observe in Figure 2.1 that on our training page, tuples with "Red Star" sellers appear only in the table "Like New Items"; thus the extraction pattern that extracts all tuples with "Red Star" had the same result as the correct pattern on the training page, but not on all other pages. A subsequent second disagreement on another page involved a somewhat similar issue. On this page, shown in Figure 2.2, there were no "Brand New Items", and thus the first table from the top was "Like New Items" and the second table was "Very Good Items". By choosing the correct set in this case, we eliminated the pattern that always extracts items from the second table from the top, and kept the pattern that extracts from any table labeled "Like New Items". Note that without a verification set, any approach that labels only a single page would likely fail on subsequent pages, since a priori both of these choices (second table from top vs. table under the string "Like New Items") are reasonable rules that make sense on certain sites.

2.3 Preliminary Discussion

We saw that in our example pages from `www.half.ebay.com`, there were a number of different possible extraction scenarios. Thus, it is not obvious at all what is a desired tuple and what is not, and a good system has to support many different scenarios. The fact that the items are organized into different tables such as "Brand New Items" or "Like New Items" attaches some meaning to the tuples that may have to be taken into account during the extraction process. However, it could be argued that maybe the extraction problem could be solved with a simpler wrapper generation system that extracts all possible tuples, followed by a filter step that eliminates any unwanted tuples, say by running an appropriate SQL query. In our example, this query would consist of two selections, one for "Red Star" and one for "Like New Items".

However, for the first condition, we would have to change the tuple definition and add an extra attribute to store, say, the file names (source URLs) of any existing star images. After using a simpler wrapper generation tool (which still has to be able to handle missing attribute values in a tuple since not all tuples have a star), we could then do a check if the file name contains "redstar.gif". Obviously, this solution requires some effort and SQL expertise from the user, while our system provides a simple solution which hides all these details from the user. For the second task, it is not easy to come up with a similar solution, since in this case the new extra attribute would have to indicate in which table the tuple appears. But this is not a trivial problem as there is only one label "Like New Items" for several tuples; we contend that our more advanced approach is more appropriate for such cases. As we saw, merely using information such as "second table from top" instead of the actual label does not work.

The first condition above also demonstrates another powerful feature of our wrapper generation system: The system can detect objects (such as images and text content) that are visually next to the tuples (or the attributes of the tuples), and utilize them as rules in the wrapper. If the data to be extracted has descriptive labels next to it, these labels can be automatically detected and utilized in the rules. This is achieved by using auxiliary properties of our page representation as discussed in Subsection 5.1.

3. CONTRIBUTIONS OF THIS PAPER

We now briefly summarize and discuss what we consider to be the main contributions of this paper. In particular:

- We describe a complete system for wrapper generation that includes an interactive interface, a powerful extraction language, and techniques for deriving and ranking extraction patterns. The system was designed in a top-down fashion, by first considering how a user should be able to interact with the system, and then deriving techniques that enable this interface.
- To implement the interface, we describe a framework based on active learning that uses an additional verification set of pages to increase the robustness of the generated wrapper without requiring the labeling of many additional examples.
- We propose the use of a category utility function [10] to rank candidate tuple sets, in order to further decrease user effort.
- We perform a detailed experimental evaluation on 14 web sites that shows that our system can capture very challenging extraction scenarios with only minimal user effort.

We now compare our approach to some other recent works; a more detailed overview of previous work on wrapper generation is provided in Section 8. We note that an active learning approach for wrapper generation was first developed by Muslea et al. in [21, 13, 23], which uses a set of pages corresponding to our verification set. Our framework can be seen as an extension of their approach. We use a much more general extraction language that captures DOM tree structure as well as visual properties of the rendered page, and group multiple such extraction patterns (hypotheses) into one wrapper corresponding to a candidate tuple set. Instead of asking a user to label additional tuples from the verification set, as in [23], we enable the user to select the correct tuple set from a ranked list.

Lixto [3, 2] also offers an interactive interface that hides most technical details. Lixto does not support a verification set, and does not provide a ranking of candidate tuple sets to decrease user effort.

Most previous work on wrapper extraction uses extraction patterns based on regular expressions on the HTML source or other expressions on the DOM tree structure, and does not consider visual properties of the rendered page. An exception is the very recent work of Zhai and Liu [30], published while our own implementation was already completed, which uses visual properties in a fully automatic wrapper generation system. As discussed, fully automatic wrapper generation is a much more challenging problem and the results are usually far less reliable, and visual properties can greatly help. Our experience shows that visual properties are also very powerful, and often essential, when capturing data sources in a semi-automatic manner.

4. SYSTEM ARCHITECTURE OVERVIEW

Our wrapper generation system consists of two main components, an *interactive user interface* that handles the user-system interactions and creates an internal representation of the web pages with auxiliary properties, and a *wrapper generation system* that constructs and tests the wrappers. An overview is shown in Figure 4.1.

The Interactive User Interface: The interface allows users to train the system on browser-displayed web pages. It is implemented within the browser using Internet Explorer browser extensions, and includes several added browser toolbar buttons. During training, the user is guided by the system with the help of browser-supported messaging tools such as alerts and confirmation messages. All interaction takes place within the browser on the exact view of the pages, and the user uses the toolbar buttons and the mouse to interact with the system. When the user hits a toolbar button, a corresponding JScript program is executed in the browser.

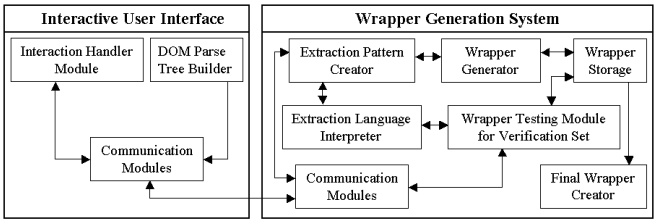


Figure 4.1: Overview of our system.

Another task of the user interface is the generation of the DOM parse tree. Instead of using tools such as JDOM [24] or Tidy [28], we generate the parse tree in the browser with the help of JScript and DHTML programming. In our experience this provides a consistent and robust way of creating DOM parse trees. Furthermore, it allows us to obtain additional useful properties from the pages, such as the visual properties of each object in the parse tree including the relative positions of objects on the rendered page. More details about the parse tree and document model are given below.

The Wrapper Generation System: The objective of this component is to generate wrappers internally, test them interactively, and release the final wrapper upon confirmation. When the DOM parse tree is created in the user interface, it is sent to this component as a stream via the communication modules. Upon receiving the stream, it is processed and stored internally in an object-oriented form. After the training example is received from the user interface, many possible extraction patterns are created in one of the key subcomponents as described in Section 6. These patterns are then run on the internal representation of the training web page to obtain the tuple sets that would be extracted. Next, we group patterns that have the same resulting tuple set on the current page into one wrapper. When the user completes the training on the training page, the selected wrapper is run on the verification set with the help of the testing module, which modifies the confirmed wrapper according to user input received from the interface. Thus, some of the extraction patterns may be removed from the wrapper. Once training is completed, the final wrapper is stored for future use. The wrapper generation system is implemented with Java Servlet Technology [29].

5. TUPLE EXTRACTION

In this section, we describe our declarative extraction language. We first present the internal representation of the web pages and their auxiliary properties. Then we describe the language in more detail and present the wrapper structure.

5.1 Document Representation

In our system, documents are represented by DOM parse trees. A DOM parse tree is an ordered tree where each node is either an element node or a text node. An element node is an internal or leaf node that has a specified tag name and tag attributes, whereas a text node is a leaf node with a single string value. This representation is created in the browser by JScript and DHTML programs that run a depth-first, left-to-right algorithm starting from the root node of the document. During this traversal, each node is marked with a unique ID, and other potentially useful information for each node is also retrieved. When the browser displays a page, the content of each node in the tree is bounded by an invisible rectangle. Using DHTML properties and methods, we can retrieve the coordinates of

this surrounding rectangle for each node, given in terms of pixels. This information is utilized by our extraction language, allowing neighboring objects on a webpage to be identified without relying on the DOM parse tree. Thus, pages are modeled as they appear in the browser.

A small portion of the DOM tree created for the eBay example is shown in Figure 5.1. The values at the upper-left corners of the nodes are the generated unique IDs. The (left, top) coordinates of the surrounding rectangles are shown in the first bracket and the (right, bottom) coordinates are shown in the second bracket. For simplicity, we do not show the tag attribute values of the element nodes in the figure. This representation is sent to the wrapper generation component where it is stored in object-oriented form.

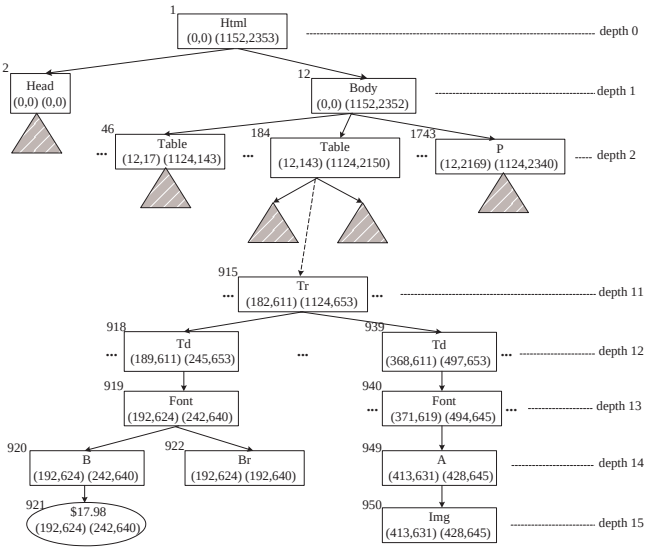


Figure 5.1: A part of the HTML DOM tree of an eBay page.

5.2 Extraction Language Overview

Our extraction language is based on the DOM tree model. Extraction rules (patterns) consist of a sequence of *expressions* that impose conditions on the path from the root to a tuple attribute. An extraction rule contains an expression for each node along this path, where each expression consists of conjunctions and disjunctions of *predicates*. In addition, wildcards can be used to skip a variable number of levels.

If a node at depth i satisfies its matching expression, then that node is considered *accepted*, otherwise *rejected*. Only children of accepted nodes at level i are checked further for the expression defined for depth $i+1$. This continues until the leaf nodes are reached, and the accepted leaf nodes form the output of the pattern on the current page. We now describe some basic types of predicates that are available; note that this is not a complete list. The language is extendable and additional predicates can be easily added.

5.3 Predicates in the Extraction Language

We first introduce some conventions used in the predicates: *N* represents a node (either an element or text node), *T* represents a tag name, *A* represents a tag attribute (name, value) pair such as A=(width, 5%), *I* represents an integer, *R* represents a regular expression which can be either a built-in expression such as "is a phone number" (or email address etc.) or a string which is created by the system on the fly, and *S* is a "combined specification" for a node (a set of (attribute, value) or (tagname, value) pairs). The extraction language assumes that children are ordered from left to right, since the document representation is created that way. The following is not an exhaustive list of the predicates, but intended to give the user a basic overview. The set of predicates may seem overly rich, but note that the user does not see any of the complexity due to our interactive framework – this in fact allows us to use such a feature-rich language.

Some Basic Predicates for Element Nodes: We list some basic predicates for element nodes, which return false if the node is not an element node.

- ***tagName(N, T)***: true iff node N has tag name T.
- ***tagAttr(N, A)***: true iff node N has a (tag attribute name, value) pair satisfying A. Note that N may also have other pairs without affecting the outcome.
- ***tagAttrArray(N, A[])***: where A is an array of (tag attribute name, value) pairs. This predicate returns true iff node N has tag attributes satisfying all the pairs in the array and contains no other additional tag attributes.
- ***elementSiblingPstn(N, I)***: true iff node N is the I-th child of its parent.
- ***childrenNumber(N, I)***: true iff node N has exactly I children.
- ***tagPstn(N, T, I)***: true iff node N has tag name T, and it is the I-th such child of its parent. This is useful when the number of left siblings varies, but the number of left siblings with tag name T remains the same.
- ***leftSibling(N, S)***: true iff node N has any left sibling which satisfies expression S. This is useful when position information alone does not work, but there is a consistently observed left sibling which identifies a relevant item. Similarly, rightSibling(N, S) and variations such as previousSibling(N, S) and nextSibling(N, S) are also available.

Basic Predicates for Text Nodes: The following predicates are defined for text nodes and return false if a node is an element node.

- ***textNode(N)***: true iff node N is a text node.
- ***textSiblingPstn(N, I)***: true iff N is the I-th child of its parent.
- ***syntax(N, R)***: true iff the string value of node N matches regular expression R. This is often used to ensure syntactic correctness of the content. There is a set of built-in regular expressions defined in the system, e.g., for dates, phone numbers, and e-mail addresses.
- ***leftTextNode(N, R, I)***: true iff the I-th closest left text neighbor node of N has a string value satisfying expression R. Note that proximity is defined in terms of distance on the rendered page, not the tree structure, so $I = 1$ tests for the closest text neighbor to the left of the current node.
- ***leftElementNode(N, S, I)***: true iff the I-th closest left leaf element node of N, according to distance on the rendered page, satisfies expression S. This predicate can match only leaf element nodes such as images (img) and break (br) nodes. Similar predicates for the other three directions are also available, as are other variations. For example, instead of stating a precise value for I, another set of predicates checks for at least one match within a range of close-by nodes.

5.4 The Wrapper Structure

During the training process multiple wrappers are generated internally and modified based on the interactions. Once training is

completed, the final wrapper is stored externally in a file. In addition to meta data such as the wrapper name, website name, and file path, a wrapper contains the following three sets of items:

(1) For each tuple attribute, a set of extraction patterns: For each tuple attribute specified in the tuple definition, the wrapper stores a set of patterns in the described extraction language. Although these patterns agree (i.e., extract the same set of items) on the training and verification pages, they may disagree on yet unencountered pages. Thus, many different patterns are stored in the wrapper, until it is inferred from the interactions that a pattern should be removed.

(2) A set of extraction patterns that define tuple regions: With the help of the previous component, attribute values can be extracted. However, given the extracted attribute values we still need to decide which of them belong to the same tuple. This is not a trivial problem due to possible missing attributes, variant attribute permutations and multiple entries that may exist in a single attribute. In general, the problem cannot be solved by simply sorting the extracted data items in the order they appear in the document and then constructing tuples based on some heuristic. In our solution, we rely on the visual representation of the web pages. We make the assumption that there exist (invisible) disjoint rectangular regions such that each region contains all attributes for one tuple. We have not seen a counterexample yet, and believe this would be unlikely due to the properties of the DOM representation. Thus, we only need to find suitable patterns that identify (extract) these regions; we refer to [12] for more details.

(3) The tuple validation rules: Our wrapper structure requires all constructed tuples to satisfy a set of *validation rules* to be output, where each validation rule is a combination of conjunctions and disjunctions of special predicates on the tuples and attributes. This component plays an important role in capturing the different extraction scenarios. The input parameters used in the predicates are as follows: *T* represents a tuple, *TA* represents an attribute of the tuple definition, *R* represents a regular expression (possibly a string value) and *E* represents an expression constructed with the predicates defined in the general extraction language. Note that E can define rules for the specification of neighbor objects. Predicates implemented in the current prototype include the following (additional predicates can be added as needed):

- ***specificAttributeValue(T, TA, R)***: true iff TA satisfies R.
- ***noMisses(T)***: true iff T has at least one item extracted for each attribute.
- ***oneItem(T)***: true iff T has at most one item extracted for each attribute.
- ***hasTheContent(T, TA, E)***: true iff TA satisfies E.

6. WRAPPER GENERATION ALGORITHM

We now give details of the wrapper generation process. To do so, we first define some internal data structures and concepts used in the algorithm. A *dom_path* object is used to represent each highlighted attribute of the training tuple in the system. In particular, a dom_path object of an attribute identifies all nodes on the path from the root to the leaf node(s). Given these dom_path objects, we find the lowest common ancestor (lca) of a training tuple, i.e., of the text nodes highlighted in the training tuple. An *LCA object* is a data structure that stores (i) a set of lca nodes for tuples on the training web page and (ii) a set of extraction patterns that extract this set of lcas. The major steps of our algorithm are as follows:

(1) Initializing the internal representations: Internal representations are created through preprocessing of the training and verification web pages.

(2) Creating dom_path and LCA objects: Upon retrieval of the training tuple, dom_path objects are created for each attribute, and multiple LCA objects are created as follows (if possible):

- Using the attribute dom_path objects, create the lca dom_path object of the training tuple.
- Using the lca dom_path object, create an extraction pattern with only tagName(N, T) predicates at each level. Execute this pattern and identify a set of lcas: This is the largest possible lca set that we allow in our system (i.e., the set of possible tuples). Create the LCA object for this largest lca set and its extraction pattern.
- Starting from the nodes in the largest lca set, move towards the root and identify the ancestor nodes at each depth.
- Starting from the root and moving down on the lca dom_object, construct expressions for each depth, where an expression is a conjunction or disjunction of predicates. The goal is to obtain expressions that accept different sets of nodes at each depth. One obvious restriction is that the node in the lca dom_path of the training tuple has to accepted.
- Create patterns by concatenating the expressions defined for each depth. Execute these patterns, and group those that extract the same set of lca nodes. Create and return LCA objects for each unique set, containing the corresponding set of extraction patterns.

(3) Creating patterns that extract tuple attributes: Given the largest lca set, we now create patterns leading from the lcas down to the attributes. The algorithm for this step is similar to the above, but a different set of predicates is used for the leaf nodes. In order to avoid creating very unlikely patterns, we apply some heuristic basic filtering rules. For example, use of neighbor predicates in an expression may be unrealistic in some cases: If a neighbor node is visually too far on the web page, or the content of it is just a single whitespace, then we do not consider such predicates for the patterns.

(4) Creating initial wrappers: For each wrapper, the attribute extraction patterns are obtained by concatenating the patterns created in (3) to the patterns stored in the LCA objects in (2). Then the patterns for extracting tuple regions are created as follows: First, the lcas of the candidate tuples are checked to see whether they are unique. If so, the set of patterns that define tuple regions is equal to the lca patterns. Otherwise, there should be some content on the web pages (such as lines, images, breaks, or some text) to visually separate the tuples. These visual separators are then identified in an iterative manner to create the tuple extraction patterns; see [12] for details.

(5) Generating the tuple validation rules and new wrappers: Using the predicates defined in Subsection 5.4, we now generate the validation rules. These rules are tested on all wrappers created so far: If a validation rule does not change the tuple set, then this rule is placed in the unconfirmed rules set of that wrapper. If a rule causes a different tuple set, then a new wrapper is replicated from that wrapper, and the rule is stored in the set of confirmed rules of the new wrapper. If this is the desired tuple set, then this means this rule must apply to future tuples.

(6) Combining the wrappers: The wrappers are analyzed, and any that generate the same tuple sets are combined.

(7) Ranking the tuple sets: This crucial step is discussed below.

(8) Getting confirmation from the user: If one of the tuple sets is confirmed, the system continues with the next step. If the user could not find the correct set, then she can now select the largest tuple set with only correct tuples, highlight an additional training tuple missing from this set, and go back to step (2).

(9) Testing the wrapper on the verification set: Once the correct tuple set is obtained with the interactions, the corresponding wrapper is tested on the web pages in the verification set to resolve any disagreements. In the case of a disagreement, we again use ranking of tuple sets as in step (7).

(10) Storing the wrapper: The final wrapper is stored in a file.

6.1 Ranking the Predicted Tuple Sets

Even though navigation of the various candidate tuple sets is fast and convenient with our interface, it is still important to order the tuple sets such that "more likely" sets are displayed first, since there can be many different tuple sets. However, it is not easy to give a general solution to this problem since (i) a user may want to extract specific information based on her own interest and needs, and (ii) as we will observe in Section 7, some websites present their data in fairly unexpected ways.

In our approach, we adopt the concept of category utility [10], which organizes data by maximizing inter-cluster dissimilarity and intra-cluster similarity. This concept was applied, e.g., in Cobweb [8], a tool for incremental clustering with categorical features that produces a partition of the input. In our application, a wrapper corresponds to a partition of the possible tuples on a page into valid and invalid tuples; it can be argued that the most interesting tuple sets are those where valid tuples are fairly similar, and invalid tuples differ significantly from the valid ones.

The goal of the *category utility function* is to maximize both the probability that two items in the same cluster have common feature value, and the probability that items from different clusters have different feature values. The category utility function is defined as

$$CU = \sum_{C} \sum_{A} \sum_{v} P(A = v)P(A = v|C)P(C|A = v), \quad (1)$$

where C is a cluster, A an attribute, and v a value. The first probability term defines the weight of the attributes used in the function. The second term is the probability that an item has value v for the attribute A, given that it belongs to cluster C. The higher this probability, the more likely it is that two items in a cluster share the same attribute values. The third term is the probability that an item belongs to cluster C, given that it has value v for attribute A. The greater this probability, the less likely it is that items from different clusters will have attribute values in common. In our ranking problem, each predicted (possible) tuple set is a partition of the possible tuples into two clusters: *tuples* and *non-tuples*. With the help of the described category utility function, each partition is scored to express the similarity among the tuples and the dissimilarity between tuples and non-tuples. In the current prototype, the attributes we use in the category utility function are:

(1) DOM Path: Whether the tuples have the same tag name and attribute paths in the DOM parse tree from leaves to root.

(2) Specific Value: Whether an attribute has the same value in all tuples.

(3) Missing Items: Whether any tuple in the set has missing items.

(4) Indexing Restriction: Whether an indexing restriction is needed to extract the predicted tuple set.

(5) Content Specification: Whether there exists a content specification (such as neighbor predicates) for all tuples in the set.

Given the vectors representing the attribute values for each predicted tuple set, the CU function returns real numbers between 0 and 1 which are then used for ranking. An interesting question is how to choose the weight term $P(A = v)$ in the CU function; we would expect the best setting to depend on the application scenario, but also more generally on the typical structure of web pages from which data is to be extracted. As a default, we assign equal weights to all attributes, which turned out to work quite well. However, we also provide an adaptive mechanism that keeps track of past decisions and updates the weight values in the CU function accordingly, possibly resulting in better ranking once the system is trained on a few (not necessarily related) scenarios.

We note that a possible alternative approach based on Rissanen's Minimum Description Length (MDL) principle [25, 26] was proposed in a very different context in [9], which introduces a system for automatically inferring a Document Type Descriptor (DTD) from a set of XML documents. While we considered such an approach as well, we do not expect it to do as well as our solution above; see [12] for more discussion.

7. EXPERIMENTAL EVALUATION

To evaluate our wrapper generation system, we conducted experiments based on data from 14 websites. Four of the data sets, Okra, BigBook, IAF (Internet Address Finder), and QS (Quote Server), are available at [19], and have been used to evaluate previous wrapper induction systems. This allows us to compare our results to previous systems such as WIEN [16], STALKER [22], and WL^2 [5]. The other ten data sets were chosen from well-known major sites:[1] AltaVista, CNN, Google, Hotjobs, IMDb, YMB (Yahoo! Message Board), MSN Q (MSN Money - Quotes), Weather, Art, and BN (Barnes and Noble). Some of these websites were already used in previous work on information extraction. We collected fifty web pages from each of these sites during July 2003; the data is available for download at `http://cis.poly.edu/~uirmak/ie`.

As described earlier, there are usually several possible tuple sets on a website, corresponding to different reasonable extraction scenarios. For the ten new data sets we chose what we considered the most natural extraction scenarios (see the above URL for more details). In the case of IMDb, where tuples are listed in multiple tables with each table containing a different filmography for the queried person, we assumed that the desired tuple set is the primary filmography of the queried actor/actress/producer, which is the first one on the page. In each experiment, we trained the system on one randomly chosen training page, and used ten other randomly selected unlabeled pages as verification set (with the exception of IAF, which only consists of 10 pages).

7.1 Initial Results on Four Web Sites

We now first report results on the four previously used data sets, to allow a comparison with other work. Table 7.1 shows the number of training tuples required by WIEN, STALKER, WL^2, and our system in order to achieve accuracies of 100%, 97%, 100% and 100%, respectively, on Okra and BigBook. (In the experiments, STALKER increases the number of training examples gradually, and stops the process when 97% accuracy is achieved.) For IAF and QS, neither WIEN not STALKER generates a successful wrapper, but WL^2 achieves 100% accuracy, as does our system. For these four data sets, our assumption for the target extraction scenario matches with that of previous wrapper generation systems. Since only very few pages were available for the IAF and QS data sets, these sites are not really perfect for our system, which benefits from larger verification sets.

We observe from Table 7.1 that for these four data sets, our system requires the user to highlight only a single training tuple to achieve 100% accuracy on the given data, outperforming the other systems. Of course, this is not a completely fair comparison since our system requires some limited additional user interaction on the verification

[1] http://www.altavista.com, http://www.cnn.com, http://www.google.com, http://hotjobs.yahoo.com, http://www.imdb.com, http://messages.yahoo.com, http://moneycentral.msn.com, http://weather.com, http://art.com, http://bn.com

	WIEN	STALKER	WL^2	Our System
Okra	46	1	1	1
BigBook	474	8	6	1
IAF	-	-	1	1
QS	-	-	4	1

Table 7.1: Number of training tuples required by our system and previous works

set to finish the training process. But this is in fact one of the main points underlying our approach: we believe that the main goal is to minimize the time and effort expended by the user, and that focusing only on the number of training examples is not the right approach. Labeling even a few additional training examples is typically significantly more time consuming than the interactions on the verification set needed in our system. Thus, we see it as one of the main strengths of our approach that a user can generate robust wrappers even for unusual tuple sets by typically only labelling one or two tuples by hand. (Of course, there may be scenarios where we only have access to prelabeled training examples, in which case an interactive approach is not appropriate.)

7.2 Results for all 14 Sites

To justify this claim, we now give more details on the exact amount of user interaction required in our system, based on results from all 14 sites. We did not attempt to design a model of user effort that weighs the various forms of interaction; ideally we would like to compare user efforts by measuring the actual times taken by human operators on the different systems (though we currently do not have access to the previous systems).

Details of the resulting user interactions on all 14 data sets are provided in Table 7.2. We again chose the training web page and the web pages in the verification set at random from all pages. Note that performance might be improved by choosing a set of very diverse web pages for the training and verification set, either through visual examination or analysis of the HTML code. During the wrapper generation process, we chose uniform weight values in the CU function for all sites and did not allow the system to update these terms. While our implementation is not optimized for computation time yet, generating wrappers and testing them on the verification set takes computation times in the order of seconds.

The columns in Table 7.2 contain (1) the total number of training examples (tuples) highlighted by the user, (2) the rank of the desired tuple set among all candidate tuple sets identified on the training page, and (3) the rank of the desired tuple set among all identified candidate tuple sets on any web page from the verification set on which an interaction occurred (if any). There were no disagreements on the verification set for 4 of the sites, and two in the case of MSN Q (both ranks are provided).

Our system extracted all tuples correctly on all test pages on all 14 sites. For 13 of the 14 websites, we did not have any disagreements on the testing set, and the generated wrappers extracted all information successfully. The system encountered a disagreement between the different extraction rules in the wrapper on the YMB website, since one variation did not occur on any page in the training or verification set. This variation was due to the appearance of a hyperlink anchor text in the body of a message. Some of the general extraction patterns were able to correctly extract all the text in the body including the anchor text, while some more specific extraction patterns extracted the message without the anchor text. Since the system ranked the former tuple set higher and continued with the more general extraction patterns (while informing the user by sending a message), we consider this a correct decision. However, recall that the user has multiple options to set the system to take appropriate actions in the case of such a disagreement.

	Highlighted Tuples	Ranking (Training)	Ranking (Verification)
Okra	1	1/3	1/2
BigBook	1	2/3	2/3
IAF	1	3/3	2/2
QS	1	7/7	1/2
AltaVista	1	1/2	2/3
CNN	1	1/1	-
Google	2	-/3 & 1/1	1/2
Hotjobs	1	1/4	-
IMDb	1	3/9	1/2
YMB	1	1/1	2/3
MSN Q	1	1/1	1/4 & 1/2
Weather	1	1/1	-
Art	1	1/32	1/2
BN	1	4/8	-

Table 7.2: Total user effort on all 14 sites.

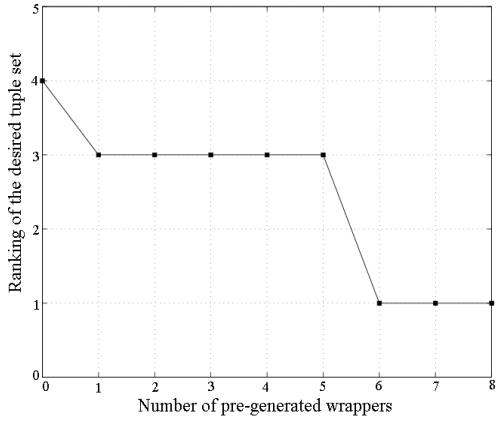
For the websites CNN, YMB, MSN Q, and Weather, the system identified only one tuple set on the training page, since there was only one tuple available on each page. In QS, a site-specific formatting decision (some values were displayed in red if there was a decrease) caused the desired tuple set to be ranked very low on the training web page. In search engines, it is common to present a result indented to the right if the previous result was from the same host. While for AltaVista this case was successfully captured and resolved on the verification set, for Google this case was not captured at that point, and thus one additional training tuple had to be supplied by the user. The reason is that our system is set up to only generate fairly general expressions at deeper depths of the DOM tree, but not at levels closer to the root, and in Google the indentation was done much closer to the root than in AltaVista. In Art, items are listed in a table of four rows and three columns (i.e., up to 12 tuples per page). Since missing attributes and many common specific values are observed, the system generated many candidate tuple sets on the training web page. However, the desired tuple set was ranked first out of 32, since all other tuple sets resulted in very small inter-cluster dissimilarity.

We do not give the time spent by the user to interact with the system, since this depends on the users and their familiarity with the system. Also not shown is the initial effort for defining the tuple structure in terms of the number of attributes and their names and types; this is the same for all systems. All interactions consist of basic tasks such as highlighting a tuple with the mouse, navigating among several identified tuple sets, or confirming a decision by clicking a button. We believe these are fairly simple tasks that can be learned within a reasonable time period by most computer user. For us, these tasks took less than two minutes total for each website.

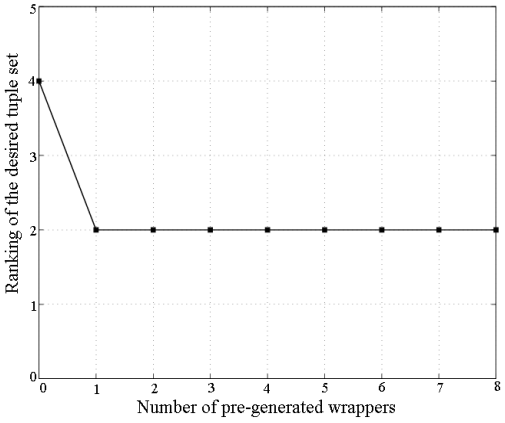
7.3 Updating Term Weights

As described in Subsection 6.1, one of our goals is to get the desired tuple set ranked high, to allow for a quick confirmation. To achieve this, we modified our system so that it can adjust its weights based on a user's past behavior. (This feature was disabled for the results in the previous subsections.) In order to evaluate the effectiveness of this adaptive approach, we conducted the following experiment: First, we examined our test platform and identified websites on which our system ranked the desired tuple sets fairly low. This was the case on BN, where the desired tuple set was ranked 4th among 8 identified tuple sets. On Art, while our system performed

well on the randomly chosen training web page in Table 7.2, we were able to find another page on which the desired tuple set would have been ranked only 4th among 29 identified tuple sets. Second, we repeatedly generated wrappers on randomly selected websites in our set (excluding BN and Art, of course, and also excluding IMDb which as mentioned had a very different extraction scenario than the others), and measured how the ranking of the desired tuple sets for BN and Art was impacted by having first trained the system on these other sites. Figure 7.1 shows the rankings of the desired tuple set on Art and BN after pre-generating wrappers on varying numbers of other websites. This result indicates that adjusting the weights may be somewhat beneficial.



(a) Art Website



(b) BN Website

Figure 7.1: The effect of pregenerating wrappers for the same extraction scenario.

Finally, we noticed that many incorrect tuple sets on Art actually had indexing restrictions. In order to observe the effect of reverse training, we used the IMDb website, which favors such extraction scenarios. We initially generated a wrapper for IMDb, and then generated a wrapper for Art. We observed that this decreased the ranking of the desired tuple in Art, and increased the ranking of some other undesirable tuple sets; however, this was corrected after subsequently training on a few other sites. (We omit the resulting figures due to space limitations.) Thus, adjusting the weights does not always result in improved ranking of the desired sets, if the system is applied to very different extraction scenarios. On the other hand, we would expect that there are some common features to many of the different extraction scenarios that arise on the web, in which case training would on average be helpful.

8. RELATED WORK

Data extraction from the web has been studied extensively over the last few years. Detailed discussions of various approaches can be found in several surveys [7, 20, 17, 15, 14]. We now discuss some of the most closely related work.

Semi-automatic wrapper induction tools such as WIEN [16], SoftMealy [11], and Stalker [22] represent documents as sequences of tokens or characters, and use machine learning techniques to induce delimiter-based extraction rules from training examples. If the web page conforms to an *HLRT organization*, WIEN can learn Header and Tail landmarks between which all data items on a page are located, plus Left and Right delimiters that mark individual items (attributes or tuples). WIEN cannot handle cases with missing items or variations in a tuple. SoftMealy [11] generates extraction rules specified as finite-state transducers. These rules can contain wildcards which allow them to overcome the above missing item problem. However, SoftMealy requires every possible case to be represented in the training examples. Stalker [22] also learns extraction rules based on landmarks, but uses a hierarchical wrapper induction algorithm with an *Embedded Catalog Tree* (ECT) formalism.

In follow-up work [21, 13, 23], an active learning approach is proposed that analyzes a set of unlabeled examples to select highly informative examples for the user to label. Our work differs from this in several ways. First, [13, 23] only consider two different types of rules: *forward* rules look at the file from start to end, while *backward* rules go from the end backwards. Our work, on the other hand, has a much more powerful extraction language and wrapper structure that captures many tricky cases. Also, our approach aims to find the desired extraction scenario directly through a series of user selections via a powerful interface, instead of asking the user to label a number of (informative) examples by hand. The system in [13], on the other hand, learns a minimal number of (perfect) disjuncts that cover all training examples.

Semi-automatic interactive wrapper induction tools that represent web pages as trees using DOM include W4F [27], XWrap [18], and Lixto [3, 2]. W4F uses a language called HEL (HTML Extraction Language) to define extraction rules. To assist the user, it offers a wizard which shows the DOM tree information for a given web page. Since the full extraction rule is programmed by the user, W4F requires expertise in HEL and HTML. XWrap [18] allows interaction between user and system via a GUI, and generates extraction rules based on certain predefined templates that limit the expressive power of the rules. The wrapper generation process takes a significant amount of time even for an expert wrapper programmer. (Each website among four sample websites required between 16 to 40 minutes in [18].) XWrap offers a testing component somewhat similar to our verification set, but requires user effort to check if the wrapper works correctly. If the tests fail, an iterative process is started that performs incremental revisions of the rules. Lixto [3, 2] generates extraction rules based on Elog, a system-internal datalog-like rule-based language. Lixto provides a sophisticated interactive user interface, and users do not have to deal with either Elog or HTML, but design their wrappers through this interface. Lixto does not provide a tools to test and train the system interactively on several web pages, but focuses on a single training page.

Work in [5] discusses the sequence-of-tokens and tree representations of web pages and proposes a system called WL^2 based on a hybrid model. WL^2 can generate successful wrappers with fewer training examples than WIEN or Stalker. One major difference between our work and WL^2 is that we allow interactions through the interface to reduce the user effort, while WL^2 requires more training examples to generate a successful wrapper.

An interactive semi-automatic tool called NoDoSe (Northwest-

ern Document Structure Extractor) [1] analyzes the structure of the documents to extract relevant data. This is achieved through a GUI where the user hierarchically decomposes the file and describes the interesting regions. NoDoSe has limited capabilities on HTML pages, but also works on plain text documents.

Fully automatic wrapper induction systems typically rely on pattern discovery techniques and are usually not reliable enough for many applications that require accurate tuple extraction. In IEPAD [4], pattern discovery techniques are applied through a data structure called a PAT tree that captures regular and repetitive patterns. RoadRunner [6] works on sample HTML pages and discovers patterns based on similarities and dissimilarities, then uses the mismatches to identify the relevant structures. Very recent independent work in [30] proposes a novel partial alignment technique to increase accuracy, and employs visual information (page layout) to identify data records. As seen in our eBay example, a user might want to only extract a specific subset of the possible tuples, and in such cases, some amount of user input is clearly needed to extract the correct set.

9. CONCLUDING REMARKS

In this paper, we have presented a new system for semi-automatic wrapper generation. Our system provides an interactive visual interface based on a new framework for generating wrappers that uses a set of verification pages. The system usually only requires one or two manually highlighted training examples to generate a reliable wrapper. This is achieved with the help of a powerful extraction language and a set of active learning and ranking techniques. We conducted experiments on multiple websites to evaluate our system; the results show that our system compares favorably to previous approaches.

10. REFERENCES

[1] B. Adelberg. NoDoSE–a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1998.

[2] R. Baumgartner, S. Flesca, and G. Gottlob. Declarative information extraction, Web crawling, and recursive wrapping with Lixto. In *Proc. of Int. Conf. on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, 2001.

[3] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *The VLDB Journal*, pages 119–128, 2001.

[4] C. Chang and S. Lui. Iepad: Information extraction based on pattern discovery. In *Proc. of the Int. World Wide Web Conf.*, 2001.

[5] W. Cohen, M. Hurst, and L. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *Proc. of the Int. World Wide Web Conf.*, 2002.

[6] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proc. of 27th Int. Conf. on Very Large Data Bases*, 2001.

[7] L. Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norwegian Computing Center, 1999.

[8] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, pages 2:139–172, 1987.

[9] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from XML documents. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2000.

[10] M. Gluck and J. Corter. Information, uncertainty, and the utility of categories. In *Proc. of 7th Annual Conf. of the Cognitive Science Society*, 1985.

[11] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[12] U. Irmak and T. Suel. Interactive wrapper generation with minimal user effort. Technical Report TR-CIS-2005-02, Polytechnic University, CIS Department, 2005.

[13] C. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and reliably extracting data from the web: A machine learning approach. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[14] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations*, 2000.

[15] S. Kuhlins and R. Tredwell. Toolkits for generating wrappers – a survey of software toolkits for automated data extraction from web sites. In *Proc. NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science (LNCS)*, 2002.

[16] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, 1997.

[17] A. Laender, B. Ribeiro-Neto, A. Silva, and J. Teixeira. A brief survey of web data extraction tools. In *SIGMOD Record*, June 2002.

[18] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *IEEE Int. Conf. on Data Engineering*, 2000.

[19] I. Muslea. RISE: Repository of online information sources used in information extraction tasks. `http://www.isi.edu/info-agents/RISE/`.

[20] I. Muslea. Extraction patterns for information extraction tasks: A survey. In *Proc. of the AAAI Workshop on Machine Learning for Information Extraction*, 1999.

[21] I. Muslea. *Active learning with multiple views*. PhD thesis, University of Southern California, 2002.

[22] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proc. of the Third Int. Conf. on Autonomous Agents (Agents'99)*, 1999.

[23] I. Muslea, S. Minton, and C. Knoblock. Active learning with strong and weak views: A case study on wrapper induction. In *Int. Joint Conf. on Artificial Intelligence*, 2003.

[24] JDOM library. `http://jdom.org/`.

[25] J. Rissanen. Modeling by shortest data description. *Automatica*, pages 14:465–471, 1978.

[26] J. Rissanen. Stochastic complexity in statistical inquiry. *World Scientific*, 1998.

[27] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *The VLDB Journal*, 1999.

[28] HTML tidy library project. `http://tidy.sourceforge.net`.

[29] Sun Microsystems, Inc. Java Servlet Technology. `http://java.sun.com/products/servlet/`.

[30] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proc. of the Int. World Wide Web Conf.*, 2005.