

**Polytechnic**  
UNIVERSITY

Brooklyn · Long Island · Westchester

# Interactive Wrapper Generation with Minimal User Effort

Utku Irmak      Torsten Suel



Department of Computer and Information  
Science

Technical Report  
TR-CIS-2005-02  
05/31/2005

# Interactive Wrapper Generation with Minimal User Effort

*Utku Irmak*      *Torsten Suel*

CIS Department  
Polytechnic University  
Brooklyn, NY 11201

uirmak@cis.poly.edu, 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 system for semi-automatic wrapper generation that can be trained on various data sources in a simple interactive manner. Our goal is to minimize the user effort for training reliable wrappers, by providing an intuitive training interface that is implemented using an underlying powerful extraction language and training algorithm. We show that our system achieves robust data extraction with significantly less user effort than previous approaches.

## 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 first extract the data embedded in the pages to make it available for further processing or storage in a relational format.

This task is usually performed by software tools called *wrappers*. The goal of a wrapper is to translate the relevant data stored in webpages into a relational (or other regular) structure for further processing. 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., [29, 22, 7, 6]. 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, say, for news items relevant to a particular stock.

In this paper, we describe a new system for semi-automatic wrapper generation. The system provides a visual interface and interactively generates a wrapper from examples specified by a human operator (user). Unlike in some previous systems, the user is not required to work on HTML source code, Document Object Model (DOM) trees, or some intermediary results generated by the system. All the interaction takes place in the browser on the original view of the webpages. The constructed wrapper is represented internally in a built-in special purpose declarative extraction language, but the user does not need to learn or understand the syntax or semantics of this language.

Our wrapper generation system employs various algorithmic techniques embedded behind the user interface. With the help of these techniques and a simple and natural user interface, the number of examples that have to be provided by hand is minimized. Usually one or two examples are enough to generate a successful wrapper. Since labeling of training data is considered the major bottleneck in wrapper generators, this approach offers great advantages. In addition, hiding all the technical details behind the user interface makes the resulting tool very easy to learn and use.

As part of our approach, we present a framework for utilizing a set of unlabeled webpages called *verification set* during the wrapper generation process. This allows us to significantly increase the accuracy and robustness of the generated wrappers through a few additional simple user interactions that require much less effort than the manual labeling of additional examples. The framework offers a solution to common obstacles to generating robust wrappers, such as missing or multiple attributes and variant attribute permutations in a tuple as described in [17]. It can also effectively cope with representational variations which may be observed for some attributes. For example, many websites present the information in various colors to indicate changes, say by showing a price in red if there was a recent decrease. This often leads to problems if a particular variation is not encountered among the limited number of labeled examples that can be provided. However, in our framework these variations can usually be handled in a fairly painless manner. As a result, the risk of later wrapper failures on unseen webpages is reduced significantly. As part of our system, we define a powerful extraction language with many built-in predicates, including some based on browser supported Dynamic HTML (DHTML) properties and methods. This language gives our system the ability to capture fairly sophisticated extraction scenarios that are often impossible to express with more limited approaches.

The rest of the paper is organized as follows. In the next section, we give an overview of the system by describing the various steps involved in our interactive wrapper generation process, and in Section 3 we discuss related work. The following three sections then describe our system in detail: First, we present the design and implementation of our system, then we describe the internal data structures and the language used to extract tuples, and finally we present the wrapper generation algorithm. Section 7 contains the experimental evaluation, and Section 8 provides some concluding remarks.

## 2 Interactive Wrapper Generation

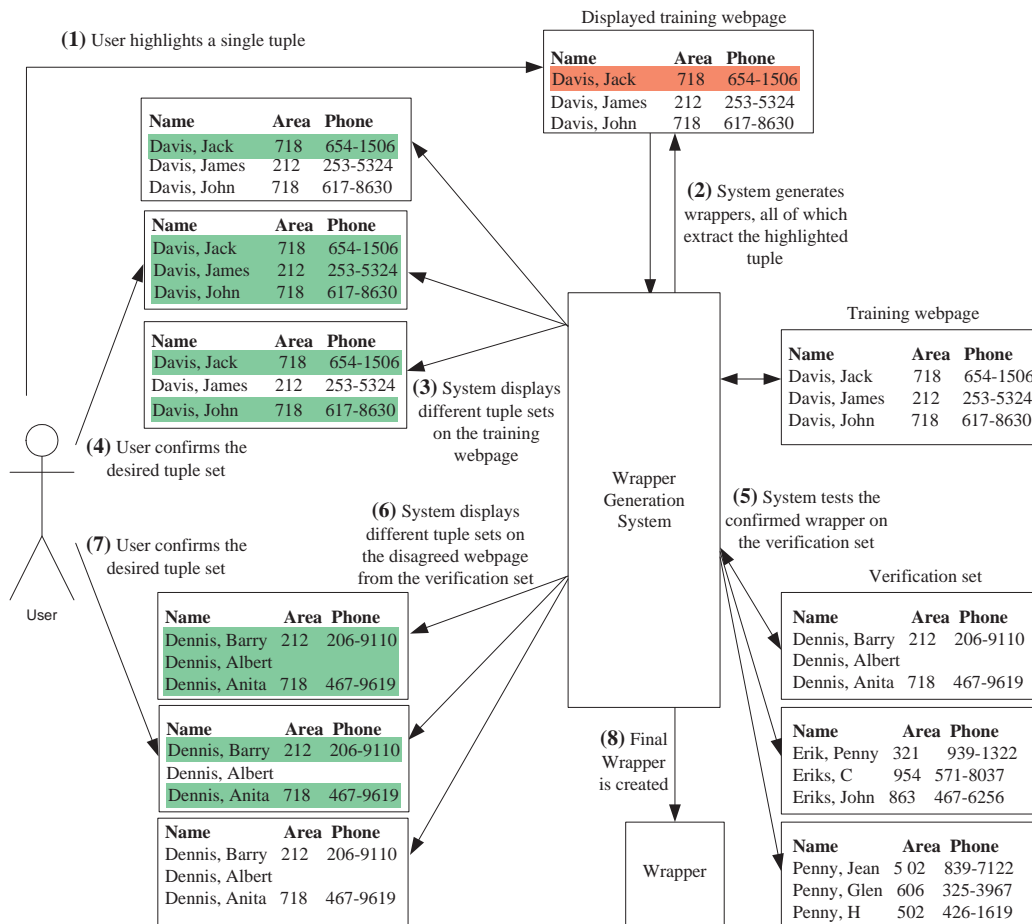
In this section, we describe the various computation and user interaction steps of our interactive wrapper generation system on some simple examples. We start with some dummy webpages to introduce the wrapper generation process and the terminology. Then we demonstrate some possible challenges faced in the extraction process and how our approach deals with these, using an example of a real world website. In the last subsection we give a discussion of the problem and possible solutions.

### 2.1 The Wrapper Generation Process in Our Approach

Our wrapper generation process is summarized in Figure 2.1. For simplicity, we show a set of artificial webpages, where each page lists the names, area codes, and phone numbers of several people.

Before describing the figure in detail, we define some terms. A *training webpage* is a page on which the user trains the system by identifying one or several tuples on the page. The *verification set* is a set of unlabeled webpages from the same website that contain data to be extracted (or pages likely to contain such data). Such pages can be obtained by visiting them using our interface, or possibly by using a simple crawler on promising parts of the website. The purpose of the verification set is discussed in detail below. Once the wrapper is constructed, it can be used on other yet unseen webpages (possibly including new pages that are created in the future) to extract the relevant information. Now we describe each step in more detail:

- (1) The user starts the training process on a training webpage, and with the system’s guidance highlights one complete tuple using the mouse, similar to [7]. In the figure, the user highlights the first tuple on the training page.
- (2) Once the tuple is entered, the system generates many *wrappers*, where each wrapper extracts a unique set of tuples on the page. Thus, the system identifies several possible tuple sets on the



**Figure 2.1:** Overview of the wrapper generation steps.

training webpage, each containing the highlighted tuple plus possibly a few other tuples.

- (3) The system then generates and displays several copies of the training webpage, where each copy highlights one of the possible tuple sets. The user can navigate between these pages using special toolbar buttons added in the browser and choose the desired tuple set. If none of them is the tuple set the user had in mind, he may try to correct one of the tuple sets, say, by adding a tuple to one of the tuple sets.
- (4) The user selects the desired tuple set among the ones shown. In the figure, the user selects the second tuple set, but note that some other possible extraction scenarios for this simple webpage could be to extract only the first tuple on a page, or to extract only those tuples which have an area code of "718". (That is, it is not always obvious what the tuples of interest are.)
- (5) Once the user has selected the desired tuple set, the system treats this as a confirmation for the corresponding wrappers. Note that a wrapper typically contains a number of different extraction

patterns, all of which generate the selected tuple set; see Subsection 5.4. (For example, all tuples, or all tuples where the name is “Davis”.) Thus, a wrapper is a set of extraction rules that are consistent on the set of data examined up to this point, but that might diverge on unseen webpages (in which case some of the rules will turn out to be incorrect). As discussed in Section 1, a possible reason are variations in the webpages, such as missing attributes and variant attribute permutations in some tuples, that may not occur in the page(s) labeled thusfar. This definition of a wrapper as a set of rules leads to the following framework to increase the accuracy and robustness of the system, where the user provides a set of additional unlabeled webpages from the same website call the *verification set*. All extraction patterns in a wrapper are then tested on the verification set to see if any of them disagree on any page. If so, then the system interacts with the user by highlighting the different tuple sets on the page with the most disagreements. When the user selects one of the tuple sets, the nonconforming patterns are removed from the wrapper and the process is repeated until there are no disagreements on the verification set. In the figure, the system tests the wrapper constructed in the previous step on the verification set and gets disagreements on all pages. The most disagreements are observed on the first page, where the wrapper outputs three different tuple sets.

- (6) The system highlights the different tuple sets and prompts the user for a choice. In the example, the first tuple set contains all available tuples on the webpage, the second only those tuples that have no missing items, and the third (empty) tuple set contains only tuples with name value “Davis”. The *validation rules* that allow extraction of tuple sets such as the last one are discussed in Section 5.
- (7) The user picks the desired tuple set from the ones displayed. The system checks again the verification set and this time no disagreements are found.
- (8) The final wrapper is stored and later used to extract data from other not yet encountered pages.

As we will show, the described framework can significantly increase the accuracy and robustness of the resulting wrapper. The total user effort is very small, typically just a few mouse clicks, and is required only if disagreements are found in the verification set. We note that the verification set itself can usually be acquired in an almost automatic fashion, by simply pointing a crawler to parts of the web site likely to contain tuples (even if only some of the retrieved pages contain tuples while others

are unrelated). If a user decides not to supply a verification set or a set that is too small, then it becomes much more likely that disagreements are encountered later in the extraction process. In the case of disagreements during the later extraction process, the system can be set up to take one of the following actions depending on the application scenario: (i) stop immediately, (ii) ignore the current page, (iii) choose the most likely extraction rule using built-in heuristics and continue to extract tuples with that wrapper, or (iv) store all different tuple sets separately to allow recovery in the future. In addition, the system can send a message to an administrator indicating potential problems and ask for a selection of the correct set as in Steps (6) and (7).

## 2.2 An Example

We now illustrate the wrapper generation process on a real webpage from `www.half.ebay.com` shown in Figure 2.2(a). A user might be interested in collecting postings of auction data from this website over a long period of time, say, for the sake of running some statistical analysis on prices, sellers, or shipping methods later on. Thus, it would be desirable to use an automated tool to extract the relevant data regularly and store it in a simple manner, say in a database. Let us assume the user is interested in extracting tuples with attributes “Price”, “TotalPrice”, “Shipping”, and “Seller” for certain products. However, in general there are multiple different reasonable extraction scenarios on these webpages, where different scenarios correspond to different user needs and lead to different tuple sets. For example, one scenario is to extract all tuples on the webpage (Figure 2.2(b)), or all tuples that appear in the table “Like New Items” (Figure 2.2(c)), or the first tuples of each table (Figure 2.2(d)) (i.e., only the cheapest items since the entries are ordered by price). Although it may not look reasonable for this webpage, another possible scenario might be to extract all tuples whose background color is gray but not white (Figure 2.2(e)). This might be necessary in cases where this color conveys some distinct meaning, as is in fact the case on certain sites.

In the rest of this section, we assume that the user is interested in extracting only tuples that appear in the table “Like New Items”, and in fact only those where the seller is awarded with a “Red Star” in “Feedback Rating” (Figure 2.2(f)). (We point out that in Figure 2.2 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 Star” represents a “Feedback Profile of 1,000 to 4,999 users”.

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(a) No tuples highlighted

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(b) All available tuples

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(c) All “Like New Items”

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(d) First tuples of each table (lowest price)

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(e) All available tuples with gray background

Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$27.50	Buy! \$30.29 (Media Mail) Upgrade	theshawdogquv (2)
\$30.00	Buy! \$32.79 (Media Mail)	mr.silver-man (239) ★
\$30.00	Buy! \$32.79 (Media Mail)	odurant (26) ★
\$44.44	Buy! \$47.23 (Media Mail)	dvinlove98 (6)

Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)
\$17.98	Buy! \$20.77 (Media Mail) Upgrade	barqainbookcompany (1711) ★
\$19.65	Buy! \$22.44 (Media Mail) Upgrade	dotcomliquid3 (2507) ★
\$23.67	Buy! \$26.46 (Media Mail)	irocknscooter (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) ★

(f) Tuples in “Like New Items” with “Red Star”

Figure 2.2: Training webpage from eBay and some possible tuple sets on this page.

We started the training process on the training webpage in Figure 2.2(a) using a verification set of size ten. When we highlighted one complete tuple on this training webpage according to the desired extraction scenario (the first tuple under “Like New Items”), the system identified 18 different possible tuple sets, including the ones in Figures 2.2(b) to (f). These sets are ranked from most to least likely according to techniques described later. We can then navigate and visually inspect the highlighted tuple sets as shown in Figures 2.2(b) to (f), which are in fact screenshots of our system. After selecting the desired tuple set (the third set in the ranking), the system tested the wrapper on the verification set, and had disagreements which were resolved with two confirmations.

The first disagreement was on a webpage which had two different tuple sets as shown in Figure 2.3(a). (Note that in Figure 2.3, the 2nd, 4th, and 6th stars from the top are red.) The tuple set in Figure 2.3(a) is the correct set, since all the tuples appear in the table “Like New Items” and all sellers are awarded with a “Red Star”. Looking again at the training webpage in Figure 2.2(a), we observe that the tuples with “Red Star” sellers appear only in the table “Like New Items”. The first wrapper that was constructed based on this page contained patterns that extract the correct tuple set as shown in Figure 2.3 (a), and also less restrictive patterns that extract tuples as shown in Figure 2.3(b), as both were consistent with the desired tuple set on the training page itself. When we confirmed the tuple set shown in Figure 2.3(a), the system removed the incorrect extraction patterns, but still required another confirmation as follows.

The second confirmation was on another webpage which again had two different tuple sets as shown in Figure 2.4. (Note that in Figure 2.4 only the 6th star from the top is red.) The empty tuple set in Figure 2.4(a) is the correct set, since there are no tuples in the table “Like New Items” whose seller has a “Red Star”. The tuple set in Figure 2.4(b) is incorrect since the single extracted tuple appears in the table “Very Good Items”. Looking at both the current page and the training webpage, we observe that in the current webpage there is no “Brand New Items” table; this changes the position (or *indexing information* of the “Like New Items” table, which is now the first table from the top while the table “Very Good Items” moves into the second position formerly occupied by “Like New Items”. The initial wrapper contained patterns that extract the correct tuple set shown in Figure 2.4(a) by looking at the actual label of the table (“Very Good Items”), as well as others that rely on position (“look at second table from top”) and that fail on the page shown in Figure 2.4. Thus, we should not expect to be able to get a robust wrapper from labeling only the initial training page in this example.

Brand New Items			Brand New Items		
Price	Total Price (Shipping)	Seller (Feedback)	Price	Total Price (Shipping)	Seller (Feedback)
\$49.99 Buy!	\$53.24 (Media Mail)	chandnan (31) ★	\$49.99 Buy!	\$53.24 (Media Mail)	chandnan (31) ★
\$53.00 Buy!	\$56.25 (Media Mail)	acva19 (4)	\$53.00 Buy!	\$56.25 (Media Mail)	acva19 (4)
\$54.55 Buy!	\$57.80 (Media Mail)	bookready (0) Upgrade	\$54.55 Buy!	\$57.80 (Media Mail)	bookready (0) Upgrade
\$54.55 Buy!	\$57.80 (Media Mail)	bookready (0) Upgrade	\$54.55 Buy!	\$57.80 (Media Mail)	bookready (0) Upgrade

Like New Items			Like New Items		
Price	Total Price (Shipping)	Seller (Feedback)	Price	Total Price (Shipping)	Seller (Feedback)
\$41.00 Buy!	\$44.25 (Media Mail) Upgrade	booksewi (1552) ★	\$41.00 Buy!	\$44.25 (Media Mail) Upgrade	booksewi (1552) ★
\$47.99 Buy!	\$51.24 (Media Mail) Upgrade	quickshipment (563) ★	\$47.99 Buy!	\$51.24 (Media Mail) Upgrade	quickshipment (563) ★
\$48.00 Buy!	\$51.25 (Media Mail)	americarwb (0)	\$48.00 Buy!	\$51.25 (Media Mail)	americarwb (0)
\$48.00 Buy!	\$51.25 (Media Mail)	shijune (0)	\$48.00 Buy!	\$51.25 (Media Mail)	shijune (0)

Very Good Items			Very Good Items		
Price	Total Price (Shipping)	Seller (Feedback)	Price	Total Price (Shipping)	Seller (Feedback)
\$43.75 Buy!	\$47.00 (Media Mail)	bunches_of_books (3933) ★	\$43.75 Buy!	\$47.00 (Media Mail)	bunches_of_books (3933) ★
\$45.84 Buy!	\$49.09 (Media Mail)	pittbul71 (31) ★	\$45.84 Buy!	\$49.09 (Media Mail)	pittbul71 (31) ★
\$50.00 Buy!	\$53.25 (Media Mail)	lbrossetti (0)	\$50.00 Buy!	\$53.25 (Media Mail)	lbrossetti (0)
\$57.35 Buy!	\$60.60 (Media Mail)	textbookxdotcom (2413) ★	\$57.35 Buy!	\$60.60 (Media Mail)	textbookxdotcom (2413) ★

(a) Correct tuple set

(b) Not all tuples appear in “Like New Items” table

Figure 2.3: First interaction on a webpage from the verification set.

### 2.3 Preliminary Discussion

We illustrated our system on pages from the eBay website, where there are many different extraction scenarios. Such different scenarios need to be supported by a good system since users may have many different objectives and pages may have many different variations. Note that on the presented webpages, the goal is to collect only those tuples that satisfy the requirements, and to discard any others. The fact that the items are categorized 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.

It could be argued that even with the fairly complicated structure of these pages and the user’s rather specialized extraction scenario, maybe the extraction problem could be solved with a less powerful wrapper generation system. A possible solution could be as follows: a simpler wrapper is generated by a less powerful wrapper generation system to extract all possible tuples from the webpages, and

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)	mfarqo (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)	kellyveichman@yahoo (9)
\$6.89 Buy!	\$10.14 (Media Mail)	boogermanie (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		
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) ★

(a) Correct: empty tuple set

(b) Incorrect: tuples must appear in “Like New Items” table

Figure 2.4: Second interaction on a webpage from the verification set.

then another application queries the resulting tuples to select the desired subset of tuples (say, by issuing a SQL query for tuples with price at most \$10, to use a very simple example). In our example, the query can be subdivided into two tasks:

- (1) Remove all tuples whose sellers do not have a “Red Star”.
- (2) Remove all tuples that do not appear in table “Like New Items”.

To accomplish the first task, we could change the tuple definition and add an extra attribute to store say the file names (source URLs) of any existing star images. Using a simpler wrapper generation tool (which still has to be able to handle missing items in a tuple), we could extract all the tuples on the webpage and then perform a selection on the file name of the image by, e.g., checking if the file name is `redstar.gif` or `yellowstar.gif`. Obviously, this solution requires some effort and SQL expertise from the user. Instead, our system proposes a built-in solution which hides any technical details from the user via a friendly interface and which requires only minimal user effort.

Note that 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. However, this is not a trivial problem as there is only one label “Like New Items” for many tuples in the table; we contend that our more advanced approach is basically needed in such cases. As we noticed, merely using indexing information such as “second table from top” instead of the actual label of the table is not adequate and fails on certain webpages.

In fact, the first task above demonstrates a very 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 with the help of auxiliary properties of our webpage representation as discussed in Subsection 5.1. To our knowledge, this is the first time these properties are being used in the wrapper generation systems. Including this feature in our framework and algorithms helps to minimize the training effort, since it can generate very expressive wrappers. Usually highlighting as few as one tuple on a single training page, and then confirming the relevant tuple set, is enough to generate a successful wrapper on many websites as we will show in Section 7.

Note that a related problem called *wrapper verification* [21] is to determine whether a wrapper is still operating correctly after it was built and deployed in an application. This is an important problem since websites may change their format and presentation at any point in time (and frequently do); in that case a wrapper may start extracting irrelevant information. One proposed solution is to compute a probabilistic similarity measure between a wrapper’s expected and observed output distribution where the similarity is a set of features such as word count, length, upper-case sensitivity, and others. Our proposed wrapper structure may in fact be quite useful in the context of the wrapper verification problem, since a wrapper often contains several extraction patterns, including some very specific ones. Thus, as the presentation and layout features of a website change, this can be detected early as some of the very specific extraction patterns may start to disagree on what the tuples are, while some of the more general ones still agree.

In the same context, one desired property of wrappers is *resilience*, i.e., the ability to continue to operate correctly in the event of presentation changes on the targeted web site [23]. We have not considered this issue in our current system and do not allow too general extraction patterns in the

wrappers. However, it would be interesting to investigate opportunities enabled by this diversity of extraction patterns contained in a wrapper, some fairly general and some very specific, that in a way each correspond to a theory about what sort of tuples the user might be interested in.

### 3 Related Work

The problem of information extraction from webpages has been studied extensively over the last few years. Detailed discussions of various approaches can be found in several surveys [11, 28, 23, 20, 19]. We now give an overview of the main approaches.

Early approaches for extracting information from webpages were mostly based on manual techniques. Thus, a human examines the HTML source and codes a wrapper, often using a special purpose language to extract the relevant information. Some well known examples for this approach are TSIMMIS [16], Minerva [9], Web-OQL[2], Florid [25] and Jedi [18].

Tools such as RAPIER [5], WHISK [34], and SRV [13] use relational learning systems to induce extraction rules. These tools can work on completely unstructured natural language text documents, since they can learn syntactic and semantic information available in the collection and employ them in the extraction rules. Typically, they are more suitable for webpages that contain free text.

Semi-automatic wrapper induction tools such as WIEN [22], SoftMealy, [17] and Stalker [29] represent documents as sequences of tokens or characters, and use machine learning techniques to induce delimiter-based extraction rules from training examples. The first tool with this approach, WIEN, learns the landmarks if the webpage conforms to an *HLRT organization* for the given extraction task. In this organization, H marks the end of the header after which the relevant data items start to occur, T marks the start of the tail behind all the data items, and L and R delimit the individual data items (attributes or tuples) between header and tail. WIEN cannot handle cases where there are missing items or variations in a tuple. SoftMealy [17] is a wrapper induction system that generates extraction rules specified as finite-state transducers. These rules can contain wildcards which allow it to overcome the above missing item problem. However, SoftMealy requires every possible case to be represented in the training examples. Stalker [29] is a supervised learning algorithm that can perform hierarchical data extraction. This is achieved through an *Embedded Catalog Tree* (ECT) formalism that also reduces the required number of training examples.

Semi-automatic wrapper induction tools that represent webpages as trees using the Document Object

Model (DOM) [8] include W4F [33], XWrap [24], and Lixto [4, 3]. W4F [33] uses a language called HEL (HTML Extraction Language) to define extraction rules, and offers a wizard to assist the user in generating wrappers. The wizard adds some invisible annotations to the given webpage, and when the user moves the cursor over some chunk of text, the wizard returns the corresponding DOM tree path. Since the wizard support is limited and the full query is programmed by the user, W4F requires expertise in HEL and HTML. XWrap [24] allows interaction between user and system with the help of a GUI, and generates extraction rules based on some 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 minutes to 40 minutes in [24].) XWrap offers a testing component similar to our verification set, but it requires user effort to check if the wrappers work correctly. If the tests are not satisfactory, an iterative process is started which performs incremental revisions of the extraction rules. Lixto [4, 3] generates extraction rules based on Elog, which is a system-internal datalog-like rule based language. Lixto provides a sophisticated visual and interactive user interface. The users do not have to deal with either Elog or HTML, but design their wrappers through this interface. Lixto does not provide a component to test and train the system interactively on several webpages, but rather focuses on a single training webpage.

Work in [7] discusses the sequence-of-tokens and tree representations of webpages and proposes a system called  $WL^2$  based on a hybrid model. This is achieved through special-purpose *builders* that can exploit different views of the document. The experimental results in [7] indicates that  $WL^2$  can generate good wrappers with fewer training examples than WIEN or Stalker. One major difference between our work and  $WL^2$  is that we allow interaction through the interface to reduce the user effort, while  $WL^2$  instead requires more training examples to generate a successful wrapper.

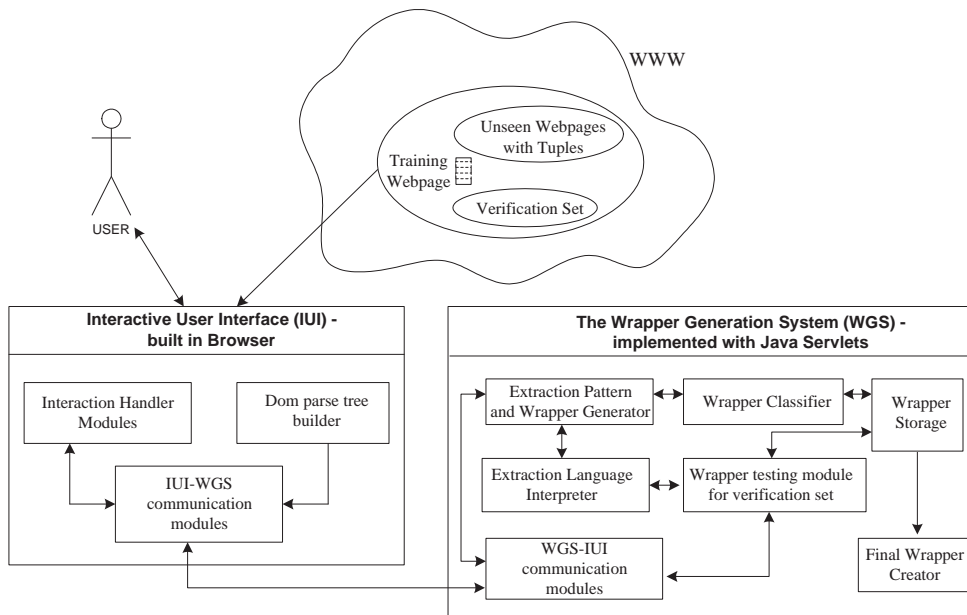
An interactive semi-automatic tool called NoDoSe (Northwestern Document Structure Extractor) [1] analyzes the structure of the documents to extract the relevant data. This is achieved through a GUI where the user hierarchically decomposes the file and describes the interesting regions. NoDoSe has limited 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 applications that require accurate tuple extraction. In IEPAD [6], pattern discovery techniques are applied through a data structure called a PAT tree that captures regular

and repetitive patterns. RoadRunner [10] works on sample HTML pages and discovers patterns based on similarities and dissimilarities, then uses the mismatches to identify the relevant structures. But as seen in the eBay example, described in Section 2, a user might want to make some refinements and only extract a specific subset of the available tuples. In such cases, some amount of user input is clearly necessary to extract the correct set of tuples.

## 4 System Overview

Our system consists of two main components. The first one is the interactive user interface which handles the user-system interaction and creates an internal representation of the webpages with auxiliary properties. The second component is the wrapper generation system which constructs the wrappers, tests them, and releases the final wrapper. An overview of the system is shown in Figure 4.1. The following subsections discuss the design and implementation of these components.



**Figure 4.1:** Overview of our system.

### 4.1 The Interactive User Interface

Our wrapper generation system offers an interactive visual interface which allows users to train the system on browser-displayed webpages. This user interface is actually a browser enhancement created using Internet Explorer browser extensions [26], with some newly added browser toolbar buttons. During the training process, 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 webpages. The user only makes use of these 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. Some of these JScript programs are created on the fly by the Wrapper Generation System through the communication modules. This is necessary since the steps taken in the training process vary between different websites, and thus the messages to guide the user can only be determined during training. In general the number of attributes in a tuple, the number of identified possible tuple sets on the training webpage, and the number of required confirmations on the verification set define the flow of the training process and are not known in advance.

In addition to interaction and communication support, another important task of the user interface is the generation of the DOM parse tree. Instead of using tools such as JDOM [30] or Tidy [35], we generate the parse tree in the browser with the help of JScript and DHTML programming techniques. Our experience has shown us that this provides a consistent and robust way of creating DOM parse trees. Furthermore, it allows us to obtain auxiliary properties on the pages, such as the visual properties of each object in the parse tree including the relative positions of objects on the page. More details about the DOM parse tree and the document model are given in Subsection 5.1.

## 4.2 The Wrapper Generation System

The main objective of this component is to generate wrappers internally, test them, and release the final wrapper upon confirmation. The *Extraction Pattern Creator* is one of the key subcomponents in the Wrapper Generation System (WGS). It is initiated by the reception of the DOM parse tree as a stream from the other component. WGS processes the stream and stores the parse tree internally in an object-oriented format. When the training example is received from the user interface, the Extraction Pattern Creator subcomponent creates many possible extraction patterns as described in Section 6.

The *Wrapper Generator and Classifier* subcomponent applies these patterns to the internal representation of the training webpage to obtain the tuple sets that would be extracted. Next, wrappers are constructed by grouping patterns that have the same resulting tuple set on the current page into one wrapper. The wrappers are then stored in the *Wrapper Storage* subcomponent. When the user completes the training on the training webpage, the *Wrapper Testing Module* is run on the verification set. This module has the privilege to modify the confirmed wrapper stored by the Wrapper Storage

subcomponent. Depending on the interactions, some refinements may be required and some of the extraction patterns may be removed from the confirmed wrapper in the Wrapper Storage. Both the Wrapper Generator and Classifier subcomponent and the Wrapper Testing Module use the *Wrapper Language Interpreter module* on the webpages stored in the verification set, since they need to apply (run) the wrappers to observe the resulting tuple sets. Both of these modules also make extensive use of the communication modules, since their flow depends on interactions with the user. Once the training is completed, the *Final Wrapper Creator Module* stores the wrapper.

The Wrapper Generation System (WGS) is implemented with Java Servlet Technology [36]. This allows flexible communication between the WGS and the browser based user interface.

## 5 Tuple Extraction

In this section, we describe our declarative wrapper language. We start with the internal representation of the webpages and their auxiliary properties, then give an overview of the language, and then describe the language in more detail. Finally, we describe the wrapper's structure and the tuple extraction process.

### 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 which 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 actually surrounded by an invisible rectangle. With the help of DHTML properties and methods, we retrieve the coordinates of this surrounding rectangles, given in terms of pixels. This information is utilized by the extraction language, allowing neighboring objects on a webpage to be identified without relying on the DOM parse tree. Thus, webpages are modeled as they appear in the browser.

A portion of the DOM tree created for the eBay example is shown in Figure 5.1. The values on 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. Since this representation is rebuilt in the Wrapper Generation System in an object-oriented structure, the properties of each node and the relations among the nodes can be accessed through the implemented methods. The Wrapper Language Interpreter module introduced in Subsection 4.2 is created based on these methods.

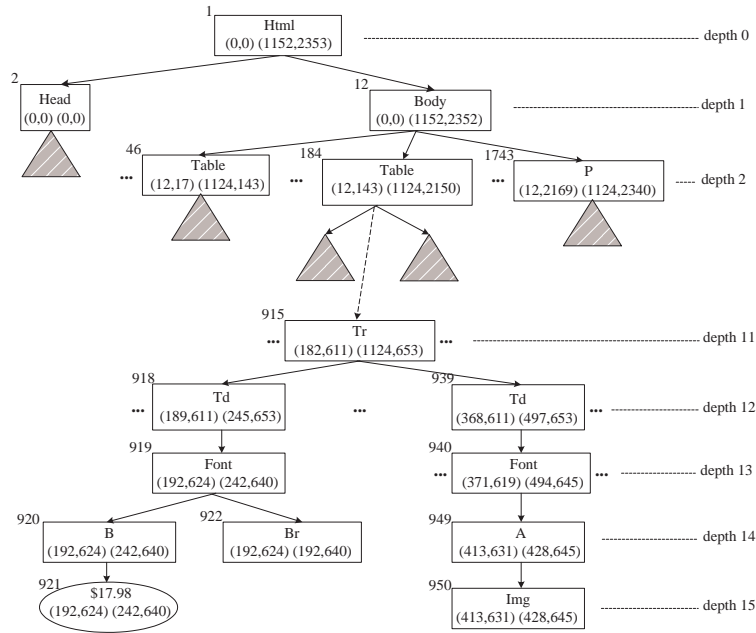


Figure 5.1: A part of the HTML DOM tree of the EBay training webpage.

## 5.2 Extraction Language Overview

Our extraction language is based on the DOM tree representation of the document. Extraction rules (patterns) are constructed in this language as a sequence of *expressions* that impose conditions on the path from the root to a valid tuple or tuple attribute. An extraction rule contains an expression for each node along this path, where an expression consists of conjunctions and disjunctions of *predicates*. In addition, wildcards could be used to skip a variable number of levels, though this is not implemented in the current prototype.

If a node at depth  $i$  satisfies its matching expression, then that node is considered *accepted*, otherwise *rejected*. Only children of accepted nodes 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. Next, we describe some basic types of predicates that are available; note that our language is extendable in that additional predicates can be easily added.

### 5.3 Predicates Used in the Extraction Language

First, we need to 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 “this is a phone number” (or email address etc.), or a string which is created by the system on the fly, and  $S$  represents a combined specification for an element node with tag name and attributes. The extraction language assumes that children are ordered from left to right, since the document representation is created that way. Note that the following is not an exhaustive list of the predicates, but intended to give the user a basic impression. The set of predicates may seem overly rich, but it has to be kept in mind that the user does not see any of the complexities due to our interactive framework, and this in fact allows us to use such a feature-rich language.

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

- $tagName(N, T)$ : returns true iff node  $N$  has tag name  $T$ .
- $tagAttr(N, A)$ : returns 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. A set of pairs required from a node can be specified in an expression with the conjunction of these predicates.
- $tagAttrArray(N, A[])$ : 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.
- $elementSiblingPstn(N, I)$ : returns true iff node  $N$  is the  $I$ -th child of its parent.
- $childrenNumber(N, I)$ : returns true iff node  $N$  has exactly  $I$  children.
- $tagPstn(N, T, I)$ : returns true iff node  $N$  has tag name  $T$ , and it is the  $I$ -th such child of its parent. This predicate is useful when the number of left siblings varies, but the number of left siblings with tag name  $T$  remains the same.
- $leftSibling(N, S)$ : returns true iff node  $N$  has any left sibling which satisfies the expression  $S$ . This is useful when position information alone is not adequate. If there is a consistently observed left sibling which happens to separate the relevant items from the others, then identify-

ing this node in the expression in essential. Similarly, `rightSibling(N, S)` and other variations such as `previousSibling(N, S)` and `nextSibling(N, S)` are also available. The latter two are more restrictive as they only check the previous and next siblings for expression `S`, respectively.

**Some Basic Predicates for Text Nodes:** The following predicates are defined for text nodes and return false if the node is not a text node.

- `textCurDepth(N)`: returns true iff node `N` is a text node at the current depth (i.e., it is not enough for some descendant to be a text node).
- `textSiblingPstn(N, I)`: returns true iff node `N` is the  $I$ -th child of its parent.
- `syntax(N, R)`: returns true iff the string value of node `N` matches the regular expression `R`. This predicate is often used to ensure syntactic correctness of the content. There is a set of built-in regular expressions defined in the system such as date, phone number, or e-mail address.
- `leftDelimiter(N, R)`: returns true iff the string value of node `N` has a left delimiter (prefix) satisfying the expression `R`. This predicate is different from `syntax(N, R)` as follows. The Extraction Language Interpreter not only searches the string for a left delimiter that satisfies `R`, but in case of a match it also outputs only the remaining substring on the right. So it serves two purposes, to eliminate items that do not have the specified left delimiter, and to perform data cleaning by removing labels or descriptive text. Similarly, `rightDelimiter(N, R)` is also available.
- `leftTextNode(N, R, I)`: returns true iff the  $I$ -th closest left text neighbor node of `N` has a string value satisfying expression `R`, where `R` can be a constant string or a regular expression. With the help of auxiliary properties stored in our document representation, it is possible to find the neighbor text nodes that are displayed visually next to node `N`. In the current prototype `R` is limited to a string constant. In the notation, the order of neighbors is defined by their distance to node `N`, so the closest left neighbor of node `N` can be tested by setting  $I = 1$ .
- `leftElementNode(N, S, I)`: returns true iff the  $I$ -th closest left leaf element node of `N` satisfies expression `S`. The predicate can match only leaf element nodes such as images (`img`) and break (`br`) nodes. Similar predicates for the other directions are also available, as are some 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 neighbor nodes.

## 5.4 The Wrapper Structure

During the training process multiple wrappers are generated internally, and modified based on the interactions. Once the training is completed, the final wrapper is stored externally. A wrapper consists of four main components which we now describe.

**(1) For each attribute, a set of extraction patterns:** For each attribute specified in the tuple definition, the wrapper stores a set of patterns in the described extraction language. When executed on the training webpage, all patterns output the same set of items (similarly on the verification webpages if available). However, as discussed earlier the patterns may disagree on yet unencountered pages. Thus, many patterns that extract the desired items are stored in the wrapper, until it is inferred from the interactions that a pattern should be removed from the set.

**(2) A set of extraction patterns that define tuple regions:** With the help of the previous component, data items for each attribute can be extracted. However, given these data items we still need to decide which of them are part of the same tuple. This is not a trivial problem and cannot be handled with a simple bottom-up approach. In other words, sorting the extracted data items in the order they appear in the document and constructing the tuples based on heuristics fails in some situations. As mentioned, common challenges in wrapper generation include coping with missing attributes, variant attribute permutations, and allowing multiple entries in a single attribute for some tuples. It is difficult to offer a heuristic-based solution capable of handling all these situations. In our solution, we also construct tuples with a bottom up approach, but we rely on the visual representation of the webpages. We make the assumption that there exist (invisible) disjoint rectangular regions such that each region contains the attributes for one unique tuple. We have not seen a counter example yet, and we believe this would be expected due to the internals of the DOM representation. Thus, our goal is to find suitable patterns that identify (extract) these regions.

There are two cases. In the first case, the system can identify element nodes whose invisible rectangles define the tuple region immediately. In the eBay example, the tuples appear in table rows, and the *tr* element nodes in the table define the tuple regions. In the second case, it may not be possible to identify such element nodes, but it is usually possible to identify some nodes which appear between the tuples, such as images, line breaks, or some text values. The top leftmost points of these nodes actually create a grid on the webpage, where the regions of the grid define the tuple regions. In

both cases, tuples are constructed by first executing the per-attribute patterns from component (1) and extracting the data items; those items that appear in the same tuple region then form a tuple.

**(3) The tuple validation rules:** The wrapper structure requires all constructed tuples to satisfy a last set of *validation rules* to be output, where each validation rule is a combination of conjunctions and disjunctions of special predicates. This component plays a significant role in defining 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 extraction language. Note that  $E$  can define rules for the specification of neighbor objects. Then the predicates used in the tuple validation rules are:

- *specificAttributeValue( $T, TA, R$ )*: returns true iff  $TA$  satisfies  $R$ .
- *tupleHasNoMissingItems( $T$ )*: returns true iff  $T$  has at least one item extracted for each attribute.
- *tupleHasOneItem( $T$ )*: returns true iff  $T$  has at most one item extracted for each attribute.
- *tupleHasTheContent( $T, TA, E$ )*: returns true iff  $TA$  satisfies  $E$ .

The set of validation rules are categorized under two different sets: *confirmed* and *unconfirmed* rules. The confirmed rules are the ones for which clear evidence is obtained during the interactions that the tuples are not desired if these rules are not satisfied. On the other hand, some rules may not be evaluated on the provided webpages, since all the constructed tuples may already satisfy these rules. This can occur if the verification set is not large enough, or if the design of the website makes sure these rules never fail. In the eBay training webpage, when the desired tuple set is confirmed, the rule which requires a red star next to the tuples will be in the confirmed rules set, since it is known that the tuple set without this rule is undesired. In other words, the rule is confirmed automatically with the confirmation of the desired tuple set. But another rule that requires no missing items in the tuples is not confirmed, since all conform to this rule so far. So this rule is placed in the unconfirmed tuple validation rules set. Additional predicates for tuple validation could be added as needed.

**(4) Internal data members:** This component stores miscellaneous data items such as the wrapper name, website name, file path, etc.

## 6 The Wrapper Generation

In this section, we describe our wrapper generation process. We first introduce the overall algorithm, and then discuss one of the steps in more detail.

### 6.1 The Wrapper Generation Algorithm

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 example (tuple) in the system. In particular, a *dom\_path* object identifies all nodes on the path from the root to the leaf (or leaves) of an attribute. Given these *dom\_path* objects, we can find the lowest common ancestor (lca) of a training example, i.e., the deepest node among the common ancestors of the text nodes highlighted in the training example. Note that on the training webpage, there can be many different lca sets, corresponding to different possible tuple sets, all of which contain the lca of the highlighted tuple. An *LCA object* is a data structure that stores a set of lca nodes and a set of extraction patterns that extract this set of lcas. An overview of our wrapper generation algorithm is given in Figure 6.1. We now describe the steps in more detail:

**(1) Initializing the internal representations (line 1):** In this step, internal representations are created through preprocessing of the training and verification webpages, as described in Subsection 5.1.

**(2) Creating *dom\_path* and LCA objects (lines 3–5):** Upon retrieval of the highlighted training example, the *dom\_path* objects are created for each attribute. Then LCA objects are created as follows:

- Examine the created attribute *dom\_path* objects to identify the lca node of the training example, and create the *dom\_path* object of the lca.
- Using the lca *dom\_path* object, create an extraction pattern with only `tagName(N, T)` predicates at each level, and execute this pattern to identify the set of lcas. This is the largest possible lca set. Create and return the LCA object containing this lca set and the extraction pattern (line 4).
- Starting from the nodes in the largest lca set, get all their ancestors. This helps to identify the candidate nodes at each depth.
- Starting from the root node, construct expressions for each depth of the tree, where an expression is a conjunction and/or disjunction of predicates. The expressions are constructed in a way that different sets of nodes are accepted at every depth, where each set contains the node

identified at the same depth of the lca dom\_path.

- Create patterns by combining the expressions defined for each depth. Execute these patterns, and identify those that extract the same lca nodes. Create and return LCA objects for each unique lca set, containing the corresponding set of extraction patterns (line 5).

```

1: init_representations(tr_webpage, verification[ ])
2: repeat
3:   tr_example  $\leftarrow$  retrieve_tuple(tr_webpage) {through interactions}
4:   LCA[0]  $\leftarrow$  init_LCA(tr_webpage, tr_example)
5:   create_LCAs(tr_webpage, LCA[ ], LCA_number)
6:   for i = 0 to attr_number - 1 do
7:     extraction_patterns[i]  $\leftarrow$  create_extraction_patterns(tr_webpage, tr_example, i, LCA[0])
8:   end for
9:   for i = 0 to LCA_number - 1 do
10:    wrapper[ ]  $\leftarrow$  init_wrapper(tr_webpage, tr_example, LCA[i], extraction_patterns[ ])
11:    identify_tuple_regions(tr_webpage, wrapper[ ])
12:    wrapper_number + = new_wrappers_created
13:   end for
14:   repeat
15:     VR  $\leftarrow$  create_validation_rule(tr_webpage, tr_example, wrapper[ ])
16:     for i = 0 to wrapper_number - 1 do
17:       if new_tuple_set(tr_webpage, i, wrapper[ ], VR) then
18:         wrapper[wrapper_number]  $\leftarrow$  fork_wrapper(wrapper[i], confirmed = VR)
19:         wrapper_number + +
20:       else
21:         update_wrapper(wrapper[ ], un_confirmed = VR)
22:       end if
23:     end for
24:   until (no_more_VRs)
25:   combine_wrappers(wrapper[ ]) {combines wrappers that generate same tuple sets}
26:   rank_tuple_sets(wrapper[ ]) {re-orders the wrappers}
27:   confirmed_wrapper  $\leftarrow$  interactions(wrapper[ ]) {tuples sets highlighted in the browser}
28: until (confirmed_wrapper != NULL)
29: confirmed_wrapper  $\leftarrow$  test_wrapper(confirmed_wrapper, verification[ ])
30: release_wrapper(confirmed_wrapper)

```

**Figure 6.1:** The wrapper generation algorithm.

**(3) Creating patterns that extract tuple attributes (lines 6–8):** Given the largest lca set and the training webpage and example, we now create the 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 are used for the leaf nodes. In order to avoid creating very unlikely patterns, we apply some basic filtering rules. For example, before using a neighbor predicate in an expression, we make sure the neighbor node is not too visually far on the webpage, or its content is not just a single whitespace, etc.

**(4) Creating initial wrappers (lines 9–13):** In this step, a set of initial wrappers is created. For each wrapper, the attribute extraction patterns are obtained by concatenating patterns created in (3) above to patterns stored in the LCA objects. Then the patterns in the second component, 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 webpages (such as lines, images, breaks, or some text) to visually separate the tuples.

Using functions similar to the neighbor predicates described in Section 5.3, the visual separators can be found as follows: The content between the last attribute of the training example and the first attribute of the next candidate tuple is scanned, and we check if this content (if exists) is repetitive on the webpage between the candidate tuples. The same procedure is repeated between the first attribute of the training example and the last attribute of the previous candidate tuple. If such separators exist, the patterns that extract them are stored in the second component of the wrapper. Although the procedure is rather simple, it usually works since this case occurs on visually simple pages. On the other hand, it is possible to create more than one wrapper for a single LCA object. Some wrappers are also eliminated internally if they generate very unlikely tuple sets. For example, if we use very specific extraction patterns for some attributes, and very general patterns for some others, we may create very unrealistic tuple sets. These can be detected and eliminated based on heuristics.

**(5) Generating the tuple validation rules and new wrappers (lines 14–24):** Using the predicates defined in Subsection 5.4, we 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. Note that if the user confirms this tuple set, then it is not possible to infer that the rule must apply to all future tuples. If the rule causes a different tuple set, then a new wrapper is replicated from that wrapper, and the rule is stored in the confirmed rules set 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 (line 25):** The wrappers are analyzed, and those that generate the same tuple sets are combined.

**(7) Ranking the tuple sets (line 26):** This step is discussed in detail in Section 6.2.

**(8) Getting confirmation from the user (line 27):** The tuple sets are presented as ranked in the previous step. If one of the tuple sets is confirmed, the system continues with the next step. If the desired

tuple set is not available, then the user is expected to confirm the largest tuple set with only correct tuples and then highlight an additional training example (back to line 3).

**(9) Testing the wrapper on the verification set (line 29):** Once the correct tuple set is obtained with the interactions, the corresponding wrapper is tested on the webpages in the verification set to resolve any disagreements.

**(10) Releasing the confirmed wrapper (line 30):** The confirmed wrapper is stored externally.

## 6.2 Ranking the Predicted Tuple Sets

Even though navigation of the various tuple sets is fast and convenient with our user 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 his own interest and needs, and (ii) as we will observe in Section 7, some websites present their content in fairly unexpected ways.

In our approach, we adopt the concept of category utility [15], which organizes data by maximizing inter-cluster dissimilarity and intra-cluster similarity. This concept was applied, e.g., in Cobweb [12], a tool for incremental clustering with categorical features that produces a partition of the input. In our application, a wrapper corresponding to a possible tuple set partitions items 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 valid and invalid tuples differ significantly in their structure or attribute values.

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 the 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 largest tuple set 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 a specific value exists for a particular attribute in all tuples.
- (3) **Missing Items:** Whether any tuple in the set has missing items.
- (4) **Indexing Restriction:** Whether any indexing restriction is needed to get the predicted tuple set.
- (5) **Content Specification:** Whether there exists a content specification (such as the use of 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 and in general 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 reasonably 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 examples. (We also allow manual modification of the weights.)

We note that an alternative solution to a similar ranking problem was proposed in a different context in [14], which introduces a system for automatically inferring a Document Type Descriptor (DTD) from a set of XML documents. In a nutshell, the system first generates candidate DTDs, then factors common subexpressions to make the candidates more concise, and finally chooses the best DTD among them. In the ranking step, a DTD is ranked based on how it balances the tradeoff between the conflicting concepts of *conciseness* and *preciseness*. The goal is to ensure the chosen DTD is succinct and also not too general, so that it captures the structure of the input sequences. This is achieved through the use of Rissanen’s Minimum Description Length (MDL) principle [31, 32].

In principle, conciseness and preciseness are also valid goals for wrappers. Extraction patterns that are too general might extract everything on the page, including the highlighted example, and also too

many things on future unseen pages. On the other hand, too specific patterns would only extract the highlighted tuple and possibly no tuples at all on any other pages, but such wrappers would typically not be very concise. Thus, a reasonable alternative approach might rank wrappers according to the number of tuples extracted and the lengths of the extraction patterns. Note that to some extent we already exploit similar ideas in our wrapper generation algorithm, by removing overly general or overly lengthy extraction patterns from consideration. It might be interesting to adapt a more formal information-theoretic framework such as the MDL principle to this part of our algorithm, though we do not expect major improvements.

## 7 The Experimental Results

To evaluate our wrapper generation system, we conducted experiments on fourteen websites listed in Table 7.1. The data sets of four of these websites, listed in Table 7.2 and available at [27], have been used to evaluate previous wrapper induction systems. This allows us to compare our results to previous systems such as WIEN [22], STALKER [29], and WL<sup>2</sup> [7]. The remaining ten websites are well-known major sites, some of which were already used in previous work on information extraction. We collected fifty webpages from each of these sites during July 2003.

Data Set Name	Source (Web Site)
Okra	RISE [27]
BigBook	RISE [27]
IAF (Internet Address Finder)	RISE [27]
QS (Quote Server)	RISE [27]
Altavista	<a href="http://www.altavista.com/">http://www.altavista.com/</a>
CNN	<a href="http://www.cnn.com/">http://www.cnn.com/</a>
Google	<a href="http://www.google.com/">http://www.google.com/</a>
Hotjobs	<a href="http://hotjobs.yahoo.com/">http://hotjobs.yahoo.com/</a>
IMDb	<a href="http://www.imdb.com">http://www.imdb.com</a>
YMB (Yahoo Message Board)	<a href="http://messages.yahoo.com/">http://messages.yahoo.com/</a>
MSN Q (MSN Moneycentral - Quotes)	<a href="http://moneycentral.msn.com/">http://moneycentral.msn.com/</a>
Weather	<a href="http://www.weather.com/">http://www.weather.com/</a>
Art	<a href="http://www.art.com/">http://www.art.com/</a>
BN (Barnes and Nobles)	<a href="http://www.bn.com/">http://www.bn.com/</a>

**Table 7.1:** Websites used for evaluating our system.

As described in Section 2, there are usually several possible tuple sets on a website, corresponding to different reasonable extraction scenarios. For consistency, we assume as default that the largest possible set is the desired tuple set on all sites, so any tuple sets created with the help of the following

tuple validation rules are considered undesired: (i) the existence of a surrounding content restriction, (ii) a specific value in a tuple field value restriction, and (iii) an indexing restriction for the desired tuple. This target extraction scenario was meaningful for all chosen websites except IMDb. On IMDb, tuples were listed in multiple tables where each table presented different filmographies for the queried person. Due to the different attributes attached to the tuple sets in different tables, 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 our experimental setup, we trained the system on one random training webpage and provide ten randomly selected unlabeled webpages as a verification set. After the wrapper was generated, we tested it on all remaining pages. In Table 7.2 we give the number of training examples (tuples) required by WIEN, STALKER, WL<sup>2</sup>, and our system in order to achieve accuracies of 100%, 97%, 100% and 100%, respectively, on Okra and Bigbook. For Internet Address Finder (IAF) and Quote Server (QS), neither WIEN nor STALKER generates a successful wrapper, but WL<sup>2</sup> achieves 100% accuracy, as does our system. For these four data sets, our assumption for the target extraction scenario (extract all tuples) 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 perfect for our system, which benefits from larger verification sets.

We observe from Table 7.2 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 all previous systems. Of course, this is not a completely fair comparison since our system requires some (usually very limited) additional user interaction on the verification 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. Labelling 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. (We note that in some scenarios, we may only have access to pre-labeled training examples, in which case our interactive approach is not appropriate.)

To justify this claim, we now give more details on the exact amount of user interaction required

	WIEN	STALKER	WL <sup>2</sup>	Our System
Okra	46	1	1	1
BigBook	474	8	6	1
IAF	-	-	1	1
QS	-	-	4	1

**Table 7.2:** Number of training tuples required by our system and previous work on four data sets.

in our system. We do 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 7.3. Before explaining and discussing these results, we mention some details of the experimental setup. We chose the training webpage and the webpages 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. All experiments were conducted on a 500MHz Pentium machine with 256MB of RAM running Windows NT 2000 and Internet Explorer 6.0 with the newly added special toolbar buttons. We did not optimize for computation time, and thus the reported CPU times are very pessimistic. We now describe each column of Table 7.3 in detail:

- (1) **Preprocessing Time:** The total preprocessing time spent on the training page and the verification set to construct the DOM representations, in minutes.
- (2) **Highlighted Tuples:** Total number of training examples (tuples) highlighted by the user.
- (3) **Wrapper Generation Time:** The computational time spent on generating possible wrappers and identifying different possible tuple sets, in seconds. In the case of Google, two tuples had to be highlighted and thus two values are reported for this and the next column.
- (4) **Ranking among All Tuple Sets (Training Webpage):** The rank of the desired tuple set among all tuple sets identified by the system on the training page.
- (5) **Verification Time:** The computational time spent testing the confirmed wrapper on the verification set, in seconds.
- (6) **Interactions:** Total number of web page interactions on the verification set.

	Preprocessing Time (mins)	Highlighted Tuples	Generation Time (secs)	Ranking (Training)	Verification Time (secs)	Interactions	Ranking (Verification)	Confirmations
Okra	1.53	1	3	1/3	11	1	1/2	2
BigBook	3.01	1	7	2/3	4	1	2/3	2
IAF	1.40	1	5	3/3	5	1	2/2	2
QS	1.27	1	18	7/7	5	1	1/2	2
Altavista	5.25	1	25	1/2	80	1	2/3	2
CNN	3.56	1	4	1/1	5	0	-	1
Google	1.68	2	3 & 3	-/3 & 1/1	5	1	1/2	3
Hotjobs	4.12	1	47	1/4	75	0	-	1
IMDb	3.84	1	15	3/9	40	1	1/2	2
YMB	3.55	1	3	1/1	5	1	2/3	2
MSN Q	2.38	1	20	1/1	165	2	1/4 & 1/2	3
Weather	5.36	1	20	1/1	110	0	-	1
Art	3.78	1	23	1/32	9	1	1/2	2
BN	5.03	1	19	4/8	55	0	-	1

**Table 7.3:** Total total user effort and CPU cost in our system.

(7) **Ranking among All Tuple Sets (Verification Set):** The rank of the desired tuple set among all the tuple sets identified on the interacted webpage from the verification set. There were two interactions in the case of MSN Q; both ranks are provided.

(8) **Confirmations:** Total number of required confirmations from the user.

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 Yahoo Message Board 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.

For the websites CNN, Yahoo Message Board, MSN Moneycentral, and Weather, the system identified only one tuple set on the training page, since there was only one tuple available on each page.

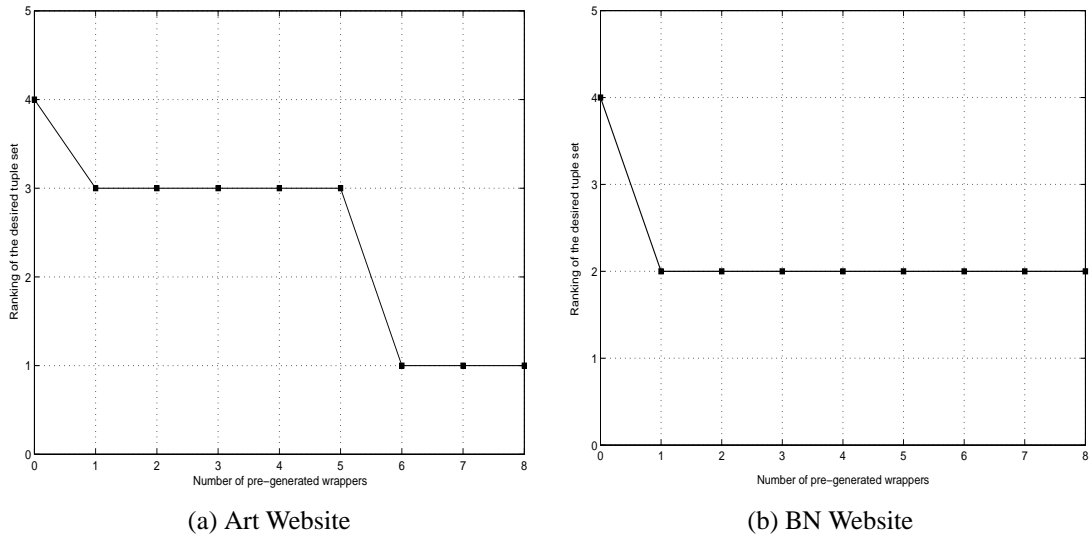
However, for MSN Moneycentral (MSN Q) two more interactions on the verification set were required to resolve disagreements. These disagreements were due to different representations of some fields in the tuples, e.g, some values were displayed in red if there was a decrease. A similar formatting issue was observed in Quote Server, causing the desired tuple set to be ranked very low (7 out of 7) on the training webpage.

In Altavista, search results were indented to the right if the previous result was from the same host. This variation did not occur on the training webpage, but did occur in the verification set. Since some extraction patterns in the confirmed wrapper were general enough to capture these tuples, this disagreement was resolved with one interaction on the verification set. Similar alignment variations were also observed in Google. When we highlighted a single tuple on the training page in Google, the system was not able to directly generate a wrapper to extract the desired tuple set. This was the case since our system was set up to only generate fairly general expressions at deeper depths of the DOM tree, but not at lower depths. In Google, the indentation was done much closer to the root of the DOM tree than in Altavista, so none of the generated extraction patterns was general enough to capture this variation. The closest tuple set proposed by the system contained only the nonindented tuples. Thus, we had to highlight one additional tuple on the training page to capture the indented tuples.

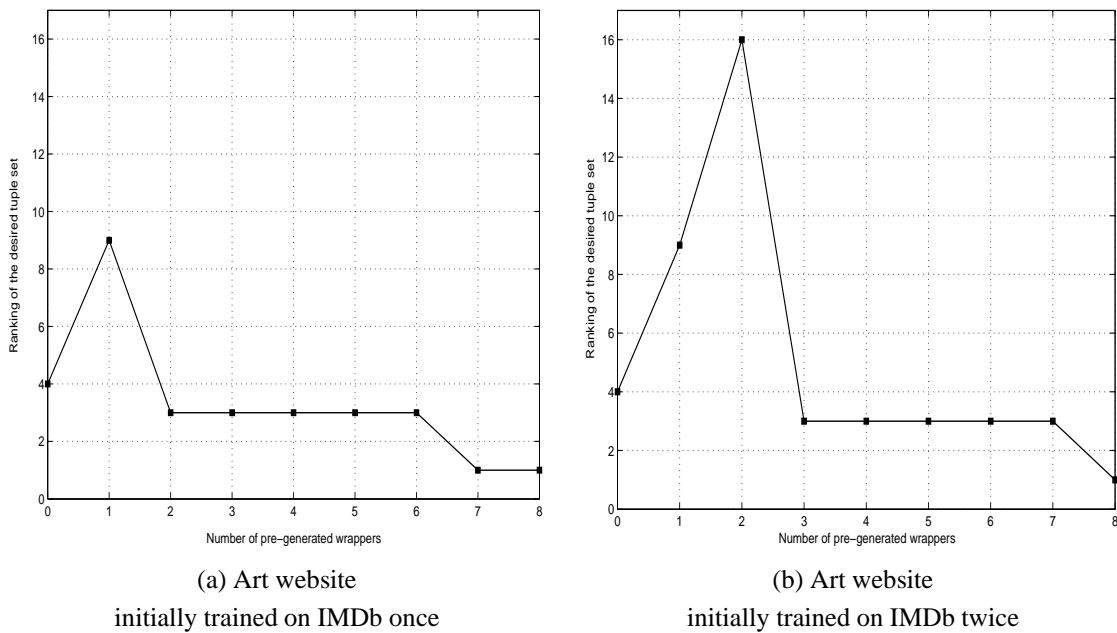
In Art, items are listed in a table of four rows and three columns (i.e., up to 12 tuples per page). Some of the tuples had missing attributes, and many tuples had common specific values for some of their fields. The system generated many different tuple sets on the training webpage based on various tuple validation rules. However, the desired tuple set was ranked first out of 32, since all other tuple sets resulted in very small inter-cluster dissimilarity.

As mentioned, the required computational time shown in Table 7.3 for preprocessing the webpages, generating possible wrappers, and testing the confirmed wrappers on the verification set are very conservative as we did not attempt to optimize CPU time. 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 (though in principle some of this could also be inferred from the page). 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.



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



**Figure 7.2:** The effect of initially training on a different extraction scenario.

As described in Subsection 6.1, one of our goals is to get the desired tuple set ranked higher to allow a quick confirmation. To achieve this, we proposed a customized system that adjusts its weights based on a user’s past behavior. (This feature was disabled for the results in the above tables.) In order to evaluate the effectiveness of this adaptive approach, we allowed our system to update the weight values in the CU function and then 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, although our system performed well on the randomly chosen training webpage, we were able to find another page on which the desired tuple set would have been ranked 4th among 29 identified tuple sets. Second, we repeatedly generated wrappers with the same target extraction scenario on different numbers of randomly selected web sites in our set (excluding BN and Art, of course, and also excluding IMDb which as mentioned had a 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.

Finally, we noticed that many incorrect tuple sets on Art actually had indexing restrictions, since the tuples were listed in a table of four rows and three columns, and that the extraction scenario in IMDb favors such restrictions. Thus, in order to observe the effect of reverse training, we initially “sabotaged” the CU function weights by generating a wrapper for IMDb with its fairly different extraction scenario once (Figure 7.2(a)) and twice (Figure 7.2(b)), and then repeated the above procedure. We see that first generating a wrapper with a different extraction scenario boosted some other tuple sets up and pulled the desired tuple set down in the ranking; however, this is corrected after subsequently training on a few sites with the same extraction scenario as the target.

## **8 Conclusions and Future Work**

In this paper, we have presented a new system for semi-automatic wrapper generation. Our system provides a visual interface based on a new framework for generating wrappers using a small verification set. The system works interactively and usually requires as few as one manually marked example to generate a successful wrapper. This is achieved with the help of algorithmic techniques and a powerful extraction language. The proposed framework allows more accurate, focused, and robust wrappers to be generated with minimal user effort. We have conducted experiments on multiple websites to evaluate our system, and the results show that our system compares favorably to previous approaches. In future work, we plan to investigate potential benefits of our approach for wrapper verification and resilience as discussed in Subsection 2.3.

**Acknowledgements:** This research was supported in part by NSF ITR Award IDM-0205647 and by the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

## References

- [1] B. Adelberg. NoDoSE—a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 283–294, 1998.
- [2] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *International Conference on Data Engineering*, pages 24–33, 1998.
- [3] R. Baumgartner, S. Flesca, and G. Gottlob. Declarative information extraction, Web crawling, and recursive wrapping with Lixto. 2001.
- [4] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with Lixto. In *The VLDB Journal*, pages 119–128, 2001.
- [5] M. E. Califf and R. J. Mooney. Relational learning of pattern-match rules for information extraction. In *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pages 6–11, Menlo Park, CA, 1998. AAAI Press.
- [6] C. Chang and S. Lui. IEPAD: Information extraction based on pattern discovery. In *Intl. World Wide Web Conf.*, 2001.
- [7] W. Cohen, M. Hurst, and L. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *Intl. World Wide Web Conf.*, 2002.
- [8] World Wide Web Consortium. The document object model. <http://www.w3.org/DOM>.
- [9] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 23(8):539–565, 1998.
- [10] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.
- [11] L. Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norwegian Computing Center, 1999.
- [12] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, pages 2:139–172, 1987.
- [13] D. Freitag. Information extraction from HTML: Application of a general machine learning approach. In *Proceedings of National Conference on Artificial Intelligence*, pages 517–523, 1998.
- [14] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from XML documents. pages 165–176, 2000.
- [15] M. Gluck and J. Corter. Information, uncertainty, and the utility of categories. In *Proceedings of 7th Annual Conference of the Cognitive Science Society*, 1985.
- [16] J. Hammer, H. García-Molina, S. Nestorov, R. Yerneni, M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system. pages 532–535, 1997.

- [17] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.
- [18] G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *Conference on Cooperative Information Systems*, pages 32–43, 1998.
- [19] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD: SIGKDD Explorations: Newsletter of the Special Interest Group (SIG) on Knowledge Discovery & Data Mining, ACM*, 2, 2000.
- [20] S. Kuhlins and R. Tredwell. Toolkits for generating wrappers – a survey of software toolkits for automated data extraction from web sites. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science (LNCS)*, pages 184–198, Berlin, October 2003. Springer.
- [21] N. Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.
- [22] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.
- [23] A. Laender, B. Ribeiro-Neto, A. Silva, and J. Teixeira. A brief survey of web data extraction tools. In *SIGMOD Record*, volume 31, June 2002.
- [24] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *International Conference on Data Engineering*, pages 611–621, 2000.
- [25] B. Ludascher, R. Himmeroder, G. Lausen, W. May, and C. Schleppehorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.
- [26] Microsoft Corporation. Browser Extensions.  
<http://msdn.microsoft.com/workshop/browser/ext/extensions.asp>.
- [27] I. Muslea. RISE: Repository of online information sources used in information extraction tasks.  
<http://www.isi.edu/info-agents/RISE/>.
- [28] I. Muslea. Extraction patterns for information extraction tasks: A survey. In *Proceedings of the AAAI Workshop on Machine Learning for Information Extraction*, pages 1–6, 1999.
- [29] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 190–197, Seattle, WA, USA, 1999. ACM Press.
- [30] JDOM Project. JDOM library. <http://jdom.org/>.
- [31] J. Rissanen. Modeling by shortest data description. *Automatica*, pages 14:465–471, 1978.
- [32] J. Rissanen. Stochastic complexity in statistical inquiry. *World Scientific*, 1998.
- [33] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *The VLDB Journal*, pages 738–741, 1999.
- [34] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.
- [35] SourceForge.net. HTML tidy library. <http://tidy.sourceforge.net>.
- [36] Sun Microsystems, Inc. Java Servlet Technology.  
<http://java.sun.com/products/servlet/>.