Browser Record and Replay as a Building Block for End-User Web Automation Tools

Sarah Chasins[1] Shaon Barman[1] Sumit Gulwani[2] Rastislav Bodik[1]

[1]University of California, Berkeley
{schasins,sbarman,bodik}@cs.berkeley.edu

[2]Microsoft Research
sumitg@microsoft.com

ABSTRACT

To build a programming by demonstration (PBD) web scraping tool for end users, one needs two central components: a list finder, and a record and replay tool. A list finder extracts logical tables from a webpage. A record and replay (R+R) system records a user's interactions with a webpage, and replays them programmatically. The research community has invested substantial work in list finding — variously called wrapper induction, structured data extraction, and template detection. In contrast, researchers largely considered the browser R+R problem solved until recently, when webpage complexity and interactivity began to rise. We argue that the increase in interactivity necessitates the use of new, more robust R+R approaches, which will facilitate the PBD web tools of the future. Because robust R+R is difficult to build and understand, we argue that tool developers need an R+R layer that they can treat as a black box.

We have designed an easy-to-use API that allows programmers to use and even customize R+R, without having to understand R+R internals. We have instantiated our API in Ringer, our robust R+R tool. We use the API to implement WebCombine, a PBD scraping tool. A WebCombine user demonstrates how to collect the first row of a relational dataset, and the tool collects all remaining rows. WebCombine uses the Ringer API to handle navigation between pages, enabling users to scrape from modern, interaction-heavy pages. We demonstrate WebCombine by collecting a 3,787,146 row dataset from Google Scholar that allows us to explore the relationship between researchers' years of experience and their papers' citation counts.

Categories and Subject Descriptors: H.5.2 [Information Interfaces and Presentation]:User Interfaces

Keywords: Record and Replay, End-User Programming, Programming by Demonstration, Automation

1. INTRODUCTION

Although the amount of data available on the Web is constantly increasing, much of this data is presented in webpages that make it difficult for end users to explore it freely. This has motivated the creation of web scraping tools designed specifically for end users.

The earliest work on PBD scraping assumed a simple workflow. First the user finds a webpage that implicitly contains a database table. He uses the PBD tool to select a small number of cells from the logical table, and the PBD tool extracts the full table. This single-page problem drove the early work on wrapper induction [2] and structured data extraction [1]. Eventually tool builders recognized the need to scrape logical tables partitioned across multiple webpages, and added support for 'Next' buttons; however, since they essentially concatenated all pages together to form a single larger page, these tools used the same core algorithms.

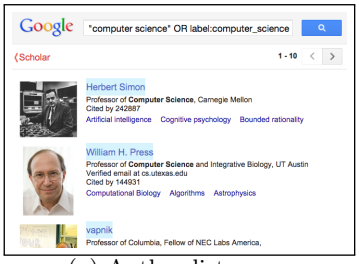
These early techniques rest on the assumption that all cells of a given row appear on a single page. For a user who wants to scrape his friends' phone numbers from Facebook, there is no such single page. Instead, from a list of friends, he must click on each friend's name to retrieve the friend's profile, then on the "About" link, and only then scrape the phone number from the "About" page. Single-page wrapper induction cannot scrape this data, because the cells in the target table are scattered across many pages.

Later, end-user web programming researchers recognized the need for an R+R layer. For instance, the mashup tool Vegemite [4] was built on top of the CoScripter [3] R+R system. Vegemite overcame the single-page restriction. It allowed users to repeat a recorded interaction for each row in a spreadsheet. Since recordings can access an arbitrary sequence of pages, users could scrape the cells of one row from multiple pages.

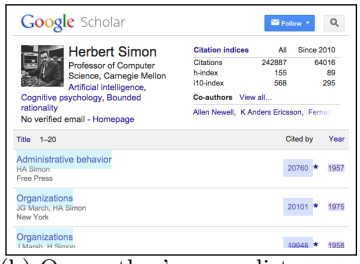
Unfortunately, the web has changed, and the replay problem has changed along with it. CoScripter — like most web R+R tools — was developed when webpages were much less interactive. Modern sites use AJAX and custom JavaScript event handlers to dynamically load information in response to user actions. While the increasing use of these technologies has made webpages more responsive and interactive, they present new challenges for standard R+R tools.

Let us consider one such challenge, an interaction that standard R+R tools cannot replay. Say a user wants to scrape the Amazon page for a given product, including the price of all options (e.g. colors). The product page shows only one price at a time, and the user must interact with the page to control which option's price is shown. To display a given option's price, the user clicks on the corresponding option button and waits as the page sends an AJAX request to retrieve the new price. The webpage remains grayed out during this time, to signal that the page is loading. The user recognizes this visual cue, and waits until the page is up-

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742849 .



(a) Author list page



(b) One author's paper list page

Herbert Simon	Administrative ...	20760	1957
Herbert Simon	Organizations	20101	1975
Herbert Simon	Organizations	19945	1958
Herbert Simon	The Sciences ...	17865	1981
...	...	...	...
William H. Press	Numerical ...	116562	1992
William H. Press	Numerical ...	113482	1992
William H. Press	Numerical ...	113289	1986
William H. Press	Numerical ...	112755	2007
...	...	...	...
vapnik	Statistical ...	55113	1998
vapnik	The Nature ...	54856	1995
vapnik	Support-vector ...	16111	1995
vapnik	A training ...	6254	1992
...	...	...	...

(c) WebCombine output

Figure 1: To use WebCombine to scrape data from Google Scholar, the user first identifies the list of authors by clicking on author names. Next the user demonstrates how to access the citations page for one author (by clicking on the link). On the resultant page, the user identifies the list of papers. WebCombine uses these demonstrations to automatically scrape the website, producing a single table of papers with each paper's author, title, citation count, and year.

dated before scraping the price, even though a price node is already present. After recording this interaction, CoScripter would issue the same events — a click on the option button and a scrape action on the price node. However, because standard R+R tools like CoScripter have no mechanism for recognizing that they must wait for new data to load, it scrapes the price before the page is updated, and thus collects the previous price. This page does contain a logical table, the list of prices, but standard R+R cannot replay the interactions required to access it.

Although the R+R problem appeared solved when earlier replay tools were introduced, today's increasingly interactive pages make traditional approaches fragile. To handle this evolution, we developed Ringer, a browser R+R tool with a new approach. Although we do not cover its implementation in this paper, at its core Ringer infers when to issue events by observing requests sent to the web server. In the Amazon example, Ringer learns that a particular AJAX response must be processed before the price can be correctly scraped. This inference step is critical to robust record and replay of interactive webpages.

Although R+R is an important building block for new PBD applications, the increase in webpage interactivity makes it unreasonable to expect programmers to build a custom R+R layer for each new tool. Even customizing existing R+R tools such as CoScripter is hardly less daunting. To address this, we designed a powerful R+R API that allows programmers to customize Ringer without having to understand its implementation. The API allows programmers to easily record interactions, treat the recordings as programs, and ultimately parameterize them. We have used this API to build WebCombine, a PBD web scraping tool.

We contribute:

- a clean interface between a robust R+R tool and applications that use it as a building block,
- a PBD scraping tool built with our API, and
- a dataset collected with our tool from Google Scholar.

2. USER INTERACTION

To motivate the WebCombine architecture, we first describe how a user interacts with WebCombine.

Assume a user wants to scrape Google Scholar to collect the data in Figure 1(c). To do this, he must 1) scrape a list of authors, 2) for each author, follow a link to the author's page, and 3) from the author's page, scrape a list of papers.

To scrape this data using WebCombine's interaction model, a user alternates between demonstrating lists (e.g. identifying a list of authors) and recording interactions (e.g. clicking on the link to an author's page). WebCombine interprets a list demonstration as the introduction of a for loop, and a recorded interaction as a procedure in the loop body. Each new loop is nested in the previous loop.

We now walk through the details of the Google Scholar example. The user starts his WebCombine script by demonstrating a list. Figure 1(a) displays a screenshot of the browser window. WebCombine's control panel is not shown. In Figure 1(a), the user has used the control panel to enter list finding mode, then clicked on the name "Herbert Simon" in the webpage. WebCombine highlights the nodes in its current hypothesized list. In this case, WebCombine has correctly hypothesized that the user wants to scrape all authors. If the list was not yet correct, the user would need to click on more sample elements. To demonstrate how to reach the next page of authors, the user will click the "Next Button" option on the control panel, then click on the arrow button in the upper right corner of the webpage. With this, the user will have completely specified the list of authors.

The user's next step is to enter recording mode. He records himself clicking on "Herbert Simon" in the browser window. This loads the author's publications page, pictured in 1(b). Once that page has opened, the user may end the recording.

In Figure 1(b), the user has started demonstrating a new list. He has clicked on the title, year, and citations of the first paper. The list finder has correctly hypothesized that the user wishes to collect the list of Herbert Simon's papers, including all clicked attributes.

Once the user finishes the process above, WebCombine generates a scraping script. Running WebCombine produces an output table, the paper citation data in Figure 1(c).

From a sequence of alternating lists and recordings — `list_1`, `recording_1`, `list_2`, ... — WebCombine creates a program of the following form:

```
for item_1 in list_1:
   table1.insert(item_1)
   parameterize recording_1 using item_1
   for item_2 in list_2:
      table_2.insert(item_2, foreign_key=item_1)
      parameterize recording_2 using item_2
      ...
```

For each for loop in the scraping program, WebCombine maintains a relational table. For this example, WebCombine would have one author table, with a single column of author names, and one paper table, with columns for title, year, and citations. Each record generated by an inner loop is linked by a foreign key to the most recent record in the immediate surrounding loop. Thus, each paper record scraped from

Function	Return Type	Description
`startRec()`	-	starts a new recording
`stopRec()`	`Program`	ends current recording
`replay(p:Program)`	`Program`	replays the program p

Table 1: The Ringer API.

Herbert Simon's page is linked to the Herbert Simon author record. Because such databases may be unfamiliar to end users, we present the data to the user as a single table by doing a natural join over all tables, as shown in Figure 1(c).

Note that the user only demonstrated the process on the first element of the outer loop, Herbert Simon. If the user had chosen to record an interaction to repeat on each paper, he would also demonstrate on only the first paper. Essentially, the user records how he collects the first row of data, leaving the tool to automatically collect the rest.

3. SYSTEM ARCHITECTURE

A WebCombine user builds a script by demonstrating lists (introducing loops) and demonstrating interactions (adding to loop bodies). These two interaction types are handled by the list finder and the R+R system respectively. In this section, we describe their functionality.

3.1 List Finder

To introduce a loop into a WebCombine program, a user must identify a list. Internally, the list finder uses a small language of node features — e.g. width, x coordinate, preceding text — to select list elements. WebCombine's task during list finding is to synthesize a selector in this language.

Although the synthesis algorithm is out of scope of this paper, the inputs to this algorithm shape the user interaction model. The inputs are a set of positive examples (DOM nodes that should be included in the list) and one of negative examples (nodes that should be excluded). During list finding, clicking on a node in the browser window adds the node to one of the two sets. After each click, the list finder generates an expression that selects all positive examples and no negative ones, then highlights all nodes selected by this expression. A click on a highlighted node adds the node to the negative examples. A click on an unhighlighted node adds it to the positive examples. Essentially, the user adjusts the list finder's hypothesis by providing counterexamples. When she is satisfied with the highlighted list, the process is done.

Users may collect lists that are partitioned across pages by identifying 'Next' or 'More' buttons, as described in Section 2. Users may also find multi-column lists, as in Figure 1(b).

3.2 R+R

To teach WebCombine to repeat an action for each element in a list, a user simply demonstrates the desired action. WebCombine must be able to 1) record the interaction, 2) replay it, and 3) replay variations on it. We have developed Ringer [5], a web R+R tool that offers this functionality.

Ringer is a record and replay system for the browser, implemented in JavaScript. It can record a user's interactions with webpages, and it can turn a recording into a script that replays the same interactions.

WebCombine treats Ringer as a black box. To make this approach possible, we had to design an R+R API that offers sufficient control to top-level applications like WebCombine.

3.2.1 The Basic R+R Interface

The basic Ringer API in Table 1 offers simple functions for starting and stopping a recording. The `stopRec` function returns a `Program` object, which top-level applications like WebCombine can store and modify as desired. They can replay them at any time with the `replay` function.

3.2.2 Parameterizing R+R Programs

In Table 2 we provide the API for parameterizing Ringer programs. The output of a Ringer recording is a straight-line program, but typically an application should not replay the same straight-line program over and over. In the case of WebCombine, we do not want to click the first element of a list 30 times. We want to click the first, then the second, and so on. Implementing this with our API is simple:

```
startRec()
p = stopRec()
p.parameterizeXPath("list_item", XPath(list[0]))
for item in list:
  p.useXpath("list_item", XPath(item))
  replay(p)
```

We identified three important parts of Ringer programs that might need to be turned into parameters. First, XPaths can be used to identify DOM nodes, so replacing XPaths allows a top-level application to direct Ringer to interact with different nodes than the ones in the user's initial demonstration. For instance, WebCombine can call a parameterized program with a sequence of different XPaths such that it clicks on each author in our list of authors, rather than clicking on Herbert Simon over and over. Second, user-typed strings may need replacing. For instance, if a user records an interaction in which he types the string "Herbert Simon email" into a search engine, WebCombine does not use that same string for each author in the list. Rather, it searches in turn for each author's name concatenated with " email". Finally, Ringer must know in which tab and in which frame of a tab to replay each event. If WebCombine is opening the author pages for all authors in a list, each in a fresh tab, it must be able to direct Ringer to the correct tab for each new author.

The Table 2 API abstracts away the details of altering event objects, adding and removing event objects, and managing Ringer's frame mapping. From the programmer's perspective, she merely replaces the values from the original demonstration with parameters to create functions, then calls the functions with arguments.

3.2.3 Customizing R+R Itself

Some applications need even more control over Ringer operation. For instance, say a user wants to scrape not only list elements, but also some of the webpage text he sees during a recording. This might arise in the author example if the user wanted to scrape the author's homepage, which appears in neither the author list nor the paper list. To address this possibility, WebCombine has a scraping mode that can be turned on and off during recording. However, Ringer has no built-in scraping mode. Without a customizable R+R layer, adding this functionality might mean modifying the internals of the R+R tool, a difficult and daunting task.

Our interface eliminates the need for such R+R hacking by allowing applications to create custom modes. The application can turn a mode on or off at any point during recording, and Ringer ensures that it is toggled at the corresponding points during replay. Applications change Ringer's behavior during a mode by associating a custom handler. While a given mode is turned on, Ringer runs the associated handler on each new event that Ringer records or replays, passing the event data object as an argument. We can implement WebCombine's scraping mode using a handler that sends the text content of clicked nodes to WebCombine. The use

Function	Description
`Program.parameterizeXPath(name:ID, origXPath:XPath)`	replace $origXPath$ with the parameter $name$
`Program.useXPath(name:ID, newXPath:XPath)`	supplies $newXPath$ as the argument for parameter $name$
`Program.parameterizeTypedString(name:ID, origString:String)`	replace $origString$ with the parameter $name$
`Program.useTypedString(name:ID, string:String)`	supplies $string$ as the argument for parameter $name$
`Program.parameterizeFrame(name: ID, origFrame: FrameID)`	replace $origFrame$ with the parameter $name$
`Program.useFrame(name:ID, frame: FrameID)`	supplies $frame$ as the argument for parameter $name$

Table 2: The API for parameterizing Ringer programs.

of modes offers a convenient way to customize Ringer's behavior without modifying its internals.

4. SYSTEM DEMONSTRATION

We used WebCombine to collect the citation counts of all papers by the 10,000 most-cited authors pursuing Computer Science research, as listed in Google Scholar [6]. Our WebCombine demonstration explores the question of when Computer Science researchers peak, based on the years in which they publish their most-cited works. With only Google Scholar data, it is difficult to draw firm conclusions about a relationship between age and impact. However, we think the data at least reveal some interesting patterns.

WebCombine is implemented as a standalone Chrome extension. Find the source and a video of this demo at `https://github.com/schasins/structured-data-scraping-extension`.

4.1 Data Collection

In the first data collection stage, we collected a set of tags. In Google Scholar, an author tagged with "Programming Languages" does not appear in a search for the "Computer Science" tag. Unfortunately, most authors are tagged with their specific subfields of interest, and hardly any with the "Computer Science" tag. Thus, collecting a good set of tags for identifying Computer Science researchers is its own scraping challenge. We leverage the fact that authors tagged with the "Computer Science" tag are often also tagged with their subfields of interest. We used WebCombine to create a loop over all the tags of all 16,979 authors in the results for the query ' "computer science" OR label:computer_science'. The result was a dataset of 51,752 tags. This included 11,439 unique tags, with frequencies ranging from 3,308 ("computer science") to 1 (e.g. "minimally cognitive agents", "misspellings", "lifestyle informatics", "yoga").

In the second stage, we scraped data about papers by the 10,000 most-cited researchers in Computer Science. To identify authors working in Computer Science, we used the tags collected in Stage 1. We sorted them by frequency, removed tags that are not primarily associated with Computer Science (e.g. "mathematics"), then selected the 65 most frequent tags. We used WebCombine to iterate over all authors with the tags, and over all those authors' papers, as detailed in Section 2. The output was a dataset of 3,787,146 paper records, each with a year and a citation count.

4.2 Data Analysis

Figure 2 reveals that, naturally, authors with short careers publish their most-cited works soon after beginning their publishing careers. However, authors with longer careers appear to remain influential well into those long careers.

In this dataset, the average author receives 26.8% of her total citations for work completed before her 10th year of publishing, 59.1% for work before her 20th year, 78.5% for work before her 30th year, and 89.7% for work before her 40th year. Highly influential authors are more likely than less influential authors to write important works later in their careers. We find that among the top 5,000 authors, 36.4% garnered more than half of their total citations from works published after the 20th year of their career. For authors in the next 5,000, the rate drops to 18.7%.

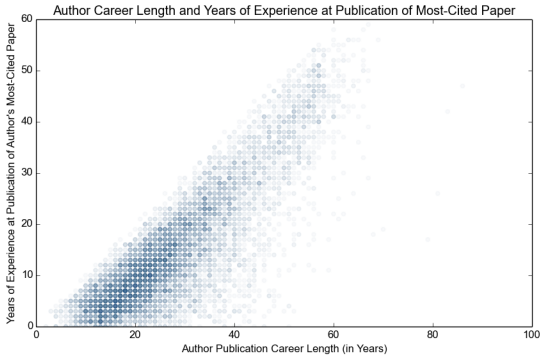


Figure 2: For each author, this graph displays the length of the author's publishing career against the years of experience when his or her most-cited paper was published.

5. CONCLUSION

We believe this demonstration constitutes preliminary evidence that R+R is a useful and usable building block for complicated end-user web automation tools. Our clean interface with a robust R+R layer allowed us to build a complex tool that empowers non-programmers to scrape large datasets, even from pages that demand user interaction. Going forward, we expect to see more and more end-user programming tools leveraging robust R+R.

6. ACKNOWLEDGEMENTS

This work is supported in part by NSF Grants CCF-1018729, CCF-1139138, CCF-1337415, and CCF-0916351, NSF Graduate Research Fellowship DGE-1106400, a grant from DOE FOA-0000619, a grant from DARPA FA8750-14-C-0011, and gifts from Mozilla, Nokia, Intel and Google.

7. REFERENCES

[1] Brad Adelberg. NoDoSE - a tool for semi-automatically extracting structured and semistructured data from text documents. In *SIGMOD Record*, 1998.

[2] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *Proc. IJCAI-97*, 1997.

[3] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. CHI '08.

[4] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. End-user programming of mashups with Vegemite. IUI '09.

[5] sbarman/webscript. `https://github.com/sbarman/webscript`.

[6] Google Scholar. Google scholar citations. `http://scholar.google.com/citations?view_op=search_authors`.