Rousillon: Scraping Distributed Hierarchical Web Data

Sarah E. Chasins
UC, Berkeley
schasins@cs.berkeley.edu

Maria Mueller
University of Washington
mm20@cs.washington.edu

Rastislav Bodik
University of Washington
bodik@cs.washington.edu

ABSTRACT

Programming by Demonstration (PBD) promises to enable data scientists to collect web data. However, in formative interviews with social scientists, we learned that current PBD tools are insufficient for many real-world web scraping tasks. The missing piece is the capability to collect hierarchically-structured data from across many different webpages. We present Rousillon, a programming system for writing complex web automation scripts by demonstration. Users demonstrate how to collect the first row of a 'universal table' view of a hierarchical dataset to teach Rousillon how to collect all rows. To offer this new demonstration model, we developed novel relation selection and generalization algorithms. In a within-subject user study on 15 computer scientists, users can write hierarchical web scrapers 8 times more quickly with Rousillon than with traditional programming.

INTRODUCTION

Web data is becoming increasingly important for data scientists [52, 34]. Social scientists in particular envision a wide range of applications driven by web data:

> "**forecasting** (e.g., of unemployment, consumption goods, tourism, festival winners and the like), **nowcasting** (obtaining relevant information much earlier than through traditional data collection techniques), **detecting health issues** and well-being (e.g. flu, malaise and ill-being during economic crises), **documenting the matching process** in various parts of individual life (e.g. jobs, partnership, shopping), and **measuring complex processes** where traditional data have known deficits (e.g. international migration, collective bargaining agreements in developing countries)." [3]

To use web datasets, data scientists must first develop web scraping programs to collect them. We conducted formative interviews with five teams of data scientists to identify design requirements for web data collection tools. The first critical requirement: do not require knowledge of HTML, DOM trees, DOM events, JavaScript, or server programming. Our formative interviews revealed that when data scientists attempt to use traditional web scraping libraries – e.g., Selenium [49], Scrapy [48], Beautiful Soup [46] – they often find themselves lacking the requisite browser expertise, especially when they need to reverse engineer browser-server communication.

UIST '18, October 14–17, 2018, Berlin, Germany

ISBN 978-1-4503-5948-1/18/10…$15.00

DOI: https://doi.org/10.1145/3242587.3242661

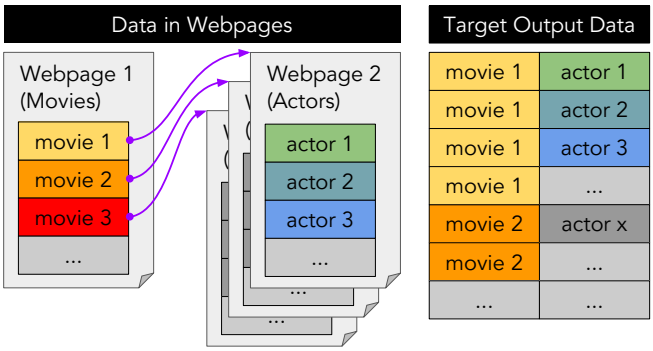


Figure 1. An example of a distributed, hierarchical web dataset. The goal is to collect a list of movies and, for each movie, a list of actors. The placement of data cells in webpages appears at left; one webpage lists multiple movies, and each movie links to a details webpage, which lists the movie's cast. The target output data appears at right; each row includes a movie and an actor who acted in the movie. The data is distributed because it appears across multiple pages: one page that lists movies and a set of movie details pages. The data is hierarchical because the root is a parent of multiple movies and each movie is a parent of multiple actors. In our formative study, 100% of data scientists' target web datasets were distributed, and 50% were hierarchical.

Programming by Demonstration (PBD) delivers on this first design requirement, offering web automation without requiring users to understand browser internals or manually reverse engineer target pages. The PBD approach has produced great successes in the web automation domain, most notably CoScripter [30], Vegemite [32], and iMacros [21], but also others [33, 50, 22, 20, 29]. CoScripter and iMacros offer record-and-replay functionality; users record themselves interacting with the browser – clicking, entering text, navigating between pages – and the tool writes a loop-free script that replays the recorded interaction. Vegemite adds a *relation extractor*, a tool for extracting tabular data from a single webpage into a spreadsheet. By putting its relation extractor and the CoScripter replayer together, Vegemite lets users users extend a data table with new columns; users can invoke a loop-free CoScripter script on each row of an extracted table, using the row cells as arguments to the script and adding the return value as a new cell. The end result is a PBD system that can collect tables of data even if cells come from multiple pages.

Our formative interviews revealed several key design requirements, but our interviewees emphasized that one in particular is critical to making a scraping tool useful and has not yet been met by prior PBD tools: collecting realistic, large-scale datasets. In particular, scrapers must handle **distributed data**, i.e., data that is dispersed across multiple webpages, and they must handle **hierarchical data**, i.e., tree-structured data.

Distributed Data

The data scientists in our formative interviews indicated that they care about *distributed data* – that is, datasets in which the

data that constitutes a single logical dataset is spread across many physical webpages. For instance, to scrape information about all movies in a box office top-10 list, we may scrape the title of each movie from the list page, then follow a link to the movie's details page to scrape the name of the movie's director, then follow a link to the director's profile page to scrape the director's bio. In this case, each of the three columns (movie title, director name, director bio) appears on a different page.

In general, web scraping includes two subtasks:

- *Single-page data extraction*: Given a page, collecting structured data from it; e.g., finding an element with a given semantic role (a title, an address), extracting a table of data.
- *Data access:* Reaching pages from which data should be extracted, either by loading new pages or causing new data to be displayed in a given page; e.g., loading a URL, clicking a link, filling and submitting a form, using a calendar widget, autocomplete widget, or other interactive component.

Many PBD scraping tools only write single-page extraction programs (e.g., Sifter [19], Solvent [41], Marmite [53], FlashExtract [29], Kimono [22], import.io [20]). A few PBD scraping tools write programs that also automate data access; in particular, record and replay tools (e.g., Ringer [4], CoScripter [30], iMacros [21]) and the Vegemite mashup tool [32] can write programs that do both extraction and data access.

Because distributed datasets split logically connected data points across many pages, a PBD tool that aims to scrape realistic data must synthesize scripts that automate data access.

Hierarchical Data

Our formative interviews also revealed that data scientists want to collect hierarchical web datasets. See Fig. 1 for an example of a hierarchical dataset. The task is to scrape, starting from a list of movies, the title of each movie, then for all actors in each movie, the name of each actor. (Note that this data is also distributed; the list of actors for each movie appears on a movie-specific page, not on the page that lists all movies.) To scrape this data, we need a program with nested loops: an outer loop to iterate through movies and an inner loop to iterate through actors for each movie.

To scrape distributed data, we need to automate data access. To scrape hierarchical data, we need nested loops. However, to date, PBD scrapers that support data access all synthesize scripts in languages that lack loops. The Ringer [4] language, iMacros language [21], and CoScripter language [30] all lack loops. Vegemite [32] uses the CoScripter language but comes closer to offering loops; by offering a UI that allows users to 'select' any subset of rows in a spreadsheet and run a loop-free CoScripter script on the selected rows, Vegemite can execute one loop, even though the output program presented to the user is loop-free. However, this approach does not extend to nested loops. One could use Vegemite to extract an initial table of movies, then run a script to extend each movie row with the top-billed actor, but could not produce the full dataset of (movie, actor) pairs depicted in Fig. 1; the script itself cannot extract or loop over tables. These tools not only synthesize programs that lack nested loops but also – because they use loop-free languages – prevent users from adding loops to the straight-line programs they output.

Introducing nested loops is a core outstanding problem in PBD. Even for domains outside of scraping, most PBD systems cannot produce them. Some, like Vegemite, lack algorithmic support (e.g., Eager [10]). Others use algorithms that could add nested loops but prevent it because it makes synthesis slow (e.g., FlashFill [14]).

PBD systems that *can* produce programs with nested loops typically require users to identify loop boundaries (e.g., SMARTedit [28]) or even whole program structures (e.g., Sketch and other template approaches [51]). This is difficult and error-prone. In the SMARTedit study, only 1 of 6 participants could add a nested loop by identifying boundaries, even though the participants were CS majors [27]. Designing interactions that enable automatic nested loop synthesis is an open problem.

To produce scraping programs with nested loops via PBD requires innovations in:

- *Algorithm design*: A synthesis algorithm that can write programs with nested loops based on a demonstration.
- *Interaction design*: An interaction model that produces demonstrations amenable to the synthesis algorithm.

This paper presents contributions in both algorithm and interaction design. In particular, we introduce a novel interaction model that makes the synthesis problem tractable. In our interaction model, a user demonstrates one row of the 'join' of their target output tables. For the Fig. 1 movie and actor data, this is the title of the first movie and name of the first actor in the first movie – the first row in the table at the right. Say we want to scrape a list of cities, for each city a list of restaurants, for each restaurant a list of reviews; we demonstrate how to collect the first city's name, the name of the first restaurant in the first city, then the first review of that first restaurant.

Given a demonstration of how to collect a row of the joined data, our custom synthesizer produces a scraping program with one loop for each level of the data hierarchy. A Relation Selector algorithm leverages common web design patterns to find relations (e.g., movies, actors, cities, restaurants, reviews). A Generalizer algorithm produces a loop for iterating over each relation. This lets our approach introduce arbitrarily nested loops without user intervention. Although our Relation Selector is a web-specific solution, designed to take advantage of web design patterns, our interaction model and Generalizer are not specialized for the web. *The success of our approach for the web scraping domain suggests a general strategy: ask the user to demonstrate one iteration of each nested loop (in our case, collect one row of the joined output table) and use domain-specific insights to identify objects that should be handled together (in our case, the rows of a given relation).*

Rousillon

This paper presents the design of Rousillon, a PBD tool that uses a novel interaction model and novel synthesis algorithms to collect distributed hierarchical data from the web. Rousillon produces a web scraping program from a single user demonstration. It is the first PBD tool that can collect hierarchical data from a tree of linked webpages.

Fig. 2 shows a usage scenario:

(a) The user opens the Rousillon browser extension.

(b) The user starts a demonstration and is asked to collect the first row of the target dataset.

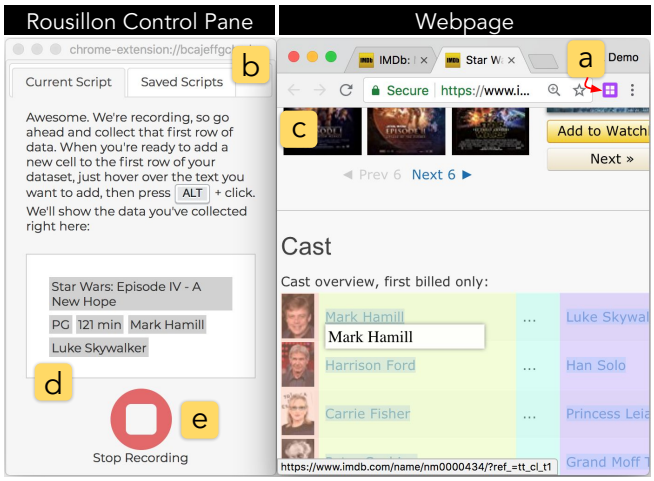


Figure 2. Using Rousillon. The user starts by clicking on (a), the icon for the Rousillon Chrome extension, which opens (b), the control pane at left. In (c), the normal browser window, the user demonstrates how to interact with pages, collect data from pages, and navigate between pages. The user collects the first row of the target dataset, which appears in the preview pane, (d). When the user clicks on (e), the 'Stop Recording' button, Rousillon synthesizes a script that collects the full dataset.

(c) In the browser window, the user demonstrates how to collect data from pages, interact with page UX elements, and navigate between pages. In this example, the user loads a webpage with a list of movies, collects the title, rating, and length of the first movie, clicks on the movie title to load a movie-specific webpage, then collects the name and role of the first actor in the movie.

(d) As the user scrapes new data cells, the cells are added to the preview box in the control panel.

(e) The user ends the demonstration by clicking on the 'Stop Recording' button. Rousillon uses the demonstration of how to scrape the first movie and the first actor of the first movie to (i) detect relations (movies, actors) and (ii) synthesize a scraping script that iterates through these two relations using the demonstrated interactions. After this process, the user can inspect and edit a blocks-based visual programming language representation of the synthesized script in the control panel.

The Rousillon architecture is depicted in Fig. 3. The input is a single user demonstration, the recording of how to collect the first row of 'joined' data. From the trace of DOM events triggered by the user's interactions, a web record-and-replay tool, Ringer [4], produces a straight-line replay script in the low-level Ringer language. From this single input, Rousillon extracts the inputs it needs for synthesizing single-page data extraction code *and* data access code. From information about the webpage elements with which the user interacted, Rousillon's **Relation Selector** identifies relations over which the user may want to iterate. Rousillon's **Reverse Compiler** translates the loop-free Ringer program into a loop-free program in a more readable language – Helena, a high-level web automation language [7]. Finally, Rousillon's **Generalizer** combines the straight-line Helena program with the relations identified by the Relation Selector and produces a Helena program with loops over the relations. This paper describes the interaction model for producing the input demonstration and the algorithms that drive Rousillon's Relation Selector, Reverse Compiler, and Generalizer.

Contributions

This paper presents the following contributions:

- A set of **design requirements for end-user web scraping tools**, discovered via formative interviews with five teams of data scientists.
- For the domain of data-access web scraping, **the first PBD tool that synthesizes programs with loops** and the first PBD tool that collects distributed hierarchical data.
- A **user study** demonstrating that programmers can write scraping programs for distributed hierarchical data 8x more quickly with PBD than with traditional programming.

RELATED WORK

We discuss related work from a broad space of web automation tools, with a special emphasis on tools that use PBD.

PBD Single-Page Data Extraction

PBD tools for data extraction typically ask users to label DOM nodes of one or more sample webpages, then produce extraction functions that take a webpage as input and produce DOM nodes as output. For instance, a web *relation extractor* takes as input a page and a subset of the page's DOM nodes that constitute part of a logical table, labeled with their row and column indexes. The output is a function that extracts a table of DOM nodes from the labeled page. Vegemite's VegeTable [32] and many industrial tools – e.g., FlashExtract [29], Kimono [22], and import.io [20] – offer PBD relation extraction.

Other PBD single-page extractors collect non-relational data. For instance, to extract product data, a user might label the product name and product price on one or more product pages. The synthesized output is a function that takes one product page as input and produces a name and price as output. Many tools offer this functionality, from the familiar Sifter [19], Solvent [41], and Marmite [53] tools to a vast body of work from the wrapper induction community [11, 5, 26, 36, 54, 25, 42, 12, 2, 24, 37, 17].

Alone, extractors cannot collect distributed data because they cannot synthesize data access. The synthesized programs do not load pages; they take one already-downloaded page as input. By definition, distributed data includes data from multiple pages, and finding and accessing those pages is part of the scraping task. Thus, while data extraction programs may be used within a distributed scraper, they cannot automate the entire collection process.

PBD Data Extraction + Data Access

Most PBD data access tools have focused on *record and replay*. A web record and replay tool or *replayer* takes as input a recording of a browser interaction and produces as output a script for replaying that same interaction. Ringer [4], Selenium Record and Playback [50], CoScripter [30], and iMacros [21] all provide this functionality, as do many others [23, 33, 18, 31, 44, 13], although some require debugging by a DOM expert and thus arguably belong outside of the PBD category.

Traditional PBD data access tools focus strictly on replaying the same loop-free interaction and thus cannot be used for large-scale web automation. The exception is the seminal Vegemite tool [32], described in the introduction. Because it can collect distributed data and generalize beyond pure replay, Vegemite comes closest of all existing PBD scraping tools

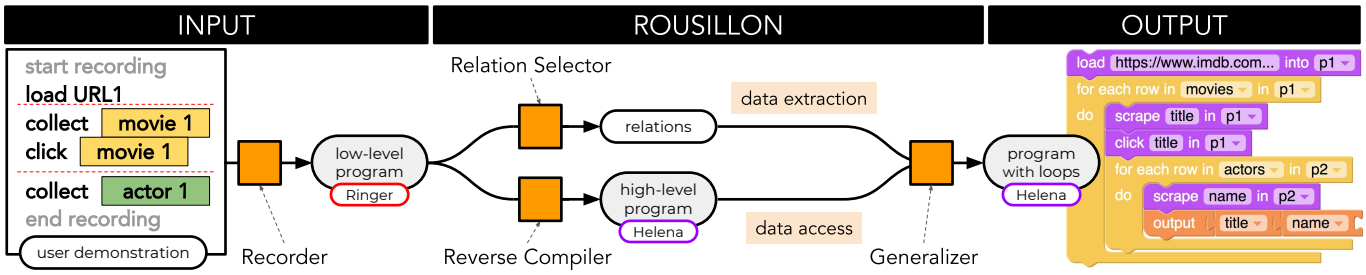


Figure 3. The Rousillon workflow. A user provides a single input demonstration, recording how to collect the first row of the target dataset. For instance, to collect the dataset depicted in Fig. 1, the user navigates to a URL that loads a list of movies, collects the first movie's title, clicks the first movie's title to load the movie's details page, collects the name of the first actor in the first movie, then ends the recording. A Recorder converts the user's demonstration into a program that replays the demonstrated interaction; it collects the first row. This replay program is the input to our PBD tool, Rousillon. Our Relation Selector uses the interacted elements (e.g., movie name, actor name) to identify relations on the target webpages; here, it finds a relation of movies on page one and actors on page two. The input replay program uses the Ringer [4] language, which is low-level and unreadable, so our Reverse Compiler translates from Ringer to the readable, high-level Helena web language. Finally, our Generalizer uses the selected relations (e.g., movies, actors) and the straight-line Helena program to write a Helena program that collects not only the first row of the data but all rows of the data.

to meeting the needs identified in our formative study. The key differences between Vegemite and Rousillon are that Vegemite:

(i) cannot add nested loops. See the introduction for a discussion of why it cannot add them, why adding them is a key technical challenge, why they are critical to scraping hierarchical data, and why hierarchical data is desirable.

(ii) uses a divided interaction model, requiring one demonstration for data extraction (relation finding) and a separate demonstration for data access (navigation). Users reported "it was confusing to use one technique to create the initial table, and another technique to add information to a new column" [32]; early versions of Rousillon used a divided interaction model and received similar feedback. Thus, Rousillon accepts a single demonstration as input and extracts both data extraction and data access information from this one input.

(iii) uses a less robust replayer, CoScripter, which was designed for an earlier, less interactive web. CoScripter's high-level language makes its programs readable but fragile in the face of page redesigns and AJAX-heavy interactive pages. On a suite of modern replay benchmarks, Ringer (the replayer Rousillon uses) replays 4x more interactions than CoScripter [4]. To use Vegemite on today's web, we would need to reimplement it with a modern replayer that uses low-level statements for robustness; thus, to recover readability, Vegemite would need a reverse compiler like Rousillon's.

(iv) can replace uses of typed strings only. E.g., Vegemite can turn a script that types "movie 1" in *node* into a script that types "movie 2" in *node*. In contrast, Rousillon can replace uses of typed strings, URLs, and DOM nodes (e.g., click on $node_2$ instead of $node_1$). The browser implementation details that make DOM node replacement more challenging than string replacement are out of scope of this paper, so although this substantially affects the possible applications of Vegemite, this paper does not emphasize this last distinction.

Another PBD data access approach uses site-provided APIs rather than webpage extraction [6]. This is a good approach if APIs offer the target data. However, this is rare in practice; none of the 10 datasets described in our formative study are available via API. (Only one dataset used a site that offers an API, and that site limits the amount of API-retrievable data to less than the team wanted and less than its webpages offer).

Web Automation Languages

There are many Domain Specific Languages (DSLs) for scraping, most implemented as libraries for general-purpose languages: Selenium [49] for C#, Groovy, Java, Perl, PHP, Python, Ruby, and Scala; Beautiful Soup [46] and Scrapy [48] for Python; Nokogiri [38] and Hpricot [16] for Ruby; HXT [15] for Haskell. Some drive a browser instance and emphasize human-like actions like clicks and keypresses; others emphasize methods for parsing downloaded DOM trees, offer no mechanisms for human-like interaction, and thus require users to reverse engineer any relevant AJAX interactions (e.g., BeautifulSoup, Scrapy). To use any of these DSLs, programmers must understand DOM trees and how to traverse them – e.g., XPath or CSS selectors – and other browser internals.

Partial PBD

While the traditional web automation DSLs described above do not use PBD, a class of GUI-wrapped DSLs do mix PBD with traditional programming. Mozenda [35] and ParseHub [43] are the best known in this class, but it also includes tools like Portia [47], Octoparse [40], and Kantu [1]. With these hybrid tools, users builds a program statement-by-statement, as in traditional programming, but they add statements via GUI menus and buttons, rather than with a text editor or mainstream structure editor. The user selects the necessary control flow constructs and other statements at each program point, as in traditional programming. However, users write node extraction code via PBD. When they reach a point in the program at which they want to use a node or table of nodes, they use the GUI to indicate that they will click on examples, and the tool writes a function for finding the relevant node or nodes. This class of tools occupies an unusual space because users need to reason about the structure of the program, the statements they will use – essentially they need to do traditional programming – but because these tools' GUIs support such small languages of actions, they do not offer the highly flexible programming models of traditional DSLs.

Rousillon Building Blocks

Rousillon makes use of two key prior works, Ringer [4] and Helena [7, 9]. **Ringer** is a web replayer. The input is a user interaction with the Chrome browser, and the output is a loop-free program in the Ringer programming language that automates the same interaction. Rousillon uses Ringer to record user demonstrations; the Ringer output program is the input to the Rousillon synthesizer. **Helena** is a high-level web automation language. With statements like `load`, `click`, and `type`, it emphasizes human-like interaction with webpages. Rousillon expresses its output programs in Helena.

FORMATIVE INTERVIEWS AND DESIGN GOALS

To explore whether current languages and tools for web automation meet data scientists' needs, we conducted formative interviews with five teams of researchers at a large U.S. university, all actively seeking web data at the time of the interviews. The teams come from a variety of disciplines. One team is comprised primarily of sociologists with two collaborators from the Department of Real Estate. All other teams were single-field teams from the following departments: Public Policy, Economics, Transportation Engineering, and Political Science. We group data collection approaches into three broad strategies: (i) automatic collection with hand-written programs, (ii) manual collection, and (iii) automatic collection with PBD-written programs. The primary focus of each interview was to explore whether a team could meet its current data needs with each strategy.

All teams considered hand-written web scraping programs out of reach, despite the fact that four of five teams had at least one team member with substantial programming experience. The Political Science team even included a programmer with web scraping experience; he had previously collected a large dataset of politicians' party affiliations using Python and Beautiful Soup [46]. He had attempted to collect the team's new target dataset with the same tools, but found his new target website made extensive use of AJAX requests, to the point where he could not reverse engineer the webpage-server communication to retrieve the target data. More commonly we saw the case where one or more team members knew how to program for a non-scraping domain – e.g. visualization, data analysis – but had attempted scraper programming without success because they lacked familiarity with DOM and browser internals. Team members found traditional web automation scripts not only unwritable but also unreadable.

In contrast, two teams considered manual data collection a viable strategy. One team went so far as to hire a high school intern to collect their data by sitting in front of the browser and copying and pasting text from webpages into a spreadsheet. Because their dataset was relatively small – only about 500 rows – this was manageable, albeit slow. Another team scaled down their target dataset to make it small enough to be collected by hand once a week, but the resultant dataset was much smaller than what they initially wanted, and the collection process still took hours of tedious copying and pasting every week. For all other teams, the target dataset was so large that they did not consider manual collection feasible.

For PBD tools, the verdict was mixed. On the one hand, all teams included at least one proficient browser user who felt comfortable demonstrating how to collect a few rows of the team's target datasets, so all teams had the skills to use them. On the other hand, most of the target collection tasks could not be expressed in the programming models of existing PBD web automation tools.

Each team had defined between one and three target datasets. Between them, the five teams had defined 10. All were distributed datasets, and all required scraping at least 500 webpages. Of the 10 target datasets, only two could be collected with existing PBD web automation tools. With a hypothetical variation on Vegemite that extends it (i) from parameterizing only typed strings to parameterizing target DOM nodes and (ii) from handling only stable input relations to handling live relation extraction across many webpages, we might collect as many as three more datasets. However, even this hypothetical improved tool would still fail to express a full half of the target datasets, because five of the 10 datasets are hierarchical.

Based on the challenges revealed in our interviews and the data scientists' stated preferences about programming style, we formulated the following design goals:

- **D1 Expertise:** Do not require knowledge of HTML, DOM trees, DOM events, JavaScript, or server programming.
- **D2 Distributed Hierarchical Data:** Handle realistic datasets. In particular, collect hierarchical data, including hierarchical data that can only be accessed via arbitrary webpage interactions and navigating between pages.
- **D3 Learnability:** Prioritize learnability by tool novices over usability by tool experts.

Guided by these goals, we designed the Rousillon PBD web scraper, which can collect all 10 of the web datasets targeted by the teams in our formative study.

ROUSILLON INTERACTION MODEL

To synthesize a scraper for distributed, hierarchical data, we need to acquire the following information from the user:

1. how to interact with and navigate to pages (data access)
2. all relations over which to iterate (data extraction)
3. the links between relations, the structure of the hierarchy

Whatever input a PBD tool takes, it must solicit all of these.

Single-Demonstration Model for Learnability. One possible approach is to require users to complete a different type of demonstration for each of the three information types. However, users find it hard to learn divided interaction models in this domain [32]. A multi-demonstration approach thus inhibits design goal D3 (prioritizing learnability). To offer a learnable interaction, PBD scrapers should aim to use only one demonstration.

Rousillon makes a contract with the user, restricting the form of the demonstrations they give, and in exchange requesting only one demonstration. In particular, we designed Rousillon to accept demonstrations in which users collect the first row of all relations in the target output data; essentially, users must demonstrate the first iteration of each loop that should appear in the final scraping program. A demonstration of this form has the potential to offer all of the requisite information types: (1) to reach the first row data, the user demonstrates all interaction and navigation actions; (2) the user touches all relations of interest; and (3) the order in which the user touches relations implicitly reveals how they are linked.

With the form of the input demonstration fixed, a few key interaction design problems arise: communicating what data users can scrape from a given element, communicating what loops users can add, and communicating how Rousillon's scraping program will operate.

Gulf of Execution. To use a PBD scraper well, users must know what data they can collect and what loops they can add. What data can be scraped from a photograph? From a canvas element? If a user interacts with the title and year of one movie in a list, will the scraper know to generalize to interacting with the titles and years of all movies? To reduce

the *gulf of execution* [39], a PBD scraper should help users answer these questions about how to use the system.

Gulf of Evaluation. Users also benefit from understanding the program a PBD scraper is building. In particular, based on the demonstration provided so far, what kind of output data will the scraper produce? Short of completing the demonstration and running the output program, how can users learn if the program will collect the data they want? To reduce the *gulf of evaluation* [39], a PBD scraper should help users answer this question about the current state of the system.

Interface

Here we describe the interface of the Rousillon PBD scraping tool and how it addresses key interaction design problems.

Implementation. Rousillon is a Chrome extension, available at `http://helena-lang.org/download` and open sourced at `https://github.com/schasins/helena`. We designed Rousillon as a Chrome extension to make installation easy and fast. Chrome extension installation is a drag-and-drop interaction, so this design choice reduces the setup and configuration complexity relative to classical web automation tools like Selenium.

Activation. The user activates the Rousillon browser extension by clicking on its icon in the URL bar (see Fig. 2(a)). Upon activation, Rousillon opens a control panel from which the user can start a new recording, load saved programs, or download data from past program runs. Because the focus of this paper is on using PBD to write scrapers, we discuss only the recording interaction for writing new programs.

Recording. Users begin by clicking a 'Start Recording' button. This opens a fresh browser window in which all the user's webpage interactions are recorded. Users are instructed to collect the cells of the first row of their target dataset.

Scraping. Users can scrape data by holding the ALT key and clicking on the webpage element they want to scrape (see Fig. 4). This indicates to Rousillon that the element's content should be the next cell in the first row of the output dataset.

Gulf of Execution. For the most part, webpages look the same during recording as in everyday browsing. However, we make two modifications to help users discover what Rousillon can do with their recorded actions, to bridge the gulf of execution.

- **What can we scrape?** Hovering over any element in a webpage displays an in-site output preview beneath the element, a hypothetical cell (see Fig. 4). This previewed cell communicates the content that Rousillon will extract if the user decides to scrape the element. The user learns what scraping an element would mean without having to write and then run a program that scrapes it. Previews are especially useful for (i) elements with text content that comes from `alt` and `title` attributes rather than displayed text and (ii) non-text elements (e.g. images are non-text elements, so Rousillon collects image source URLs).
- **What loops can we add?** Hovering over any element that appears in a known relation highlights all elements of the known relation (see Fig. 5). This emphasis of relation structure communicates that Rousillon knows how to repeat interactions performed on a row of the relation for all rows of the relation; it gives users additional control over the PBD process by indicating they can interact with a given set of elements to add a given loop in the output program.

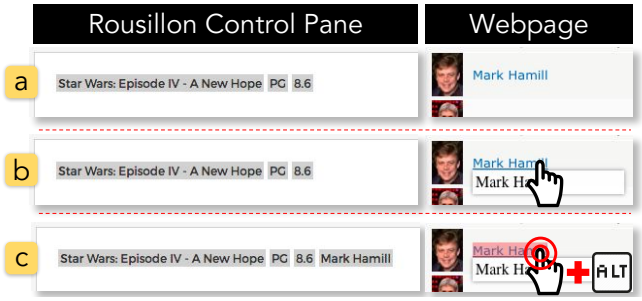


Figure 4. Scraping data. Boxes on the left show snippets of the Rousillon control pane at various points during demonstration. Boxes on the right show snippets of the webpage with which the user is interacting, at the same points in time. a) The user has already added a movie title, PG rating, and star rating to the first row of data, shown in the first row preview at left. In the webpage, we see an actor's picture and link, but the user is not hovering over them. b) An in-site output preview. When the user hovers over an element, an inline preview shows a cell the user could add to the first row. Here the user hovers over the actor name. c) The user holds the ALT key and clicks on the actor name. The text from the preview cell is added to the first row of data (see preview at left).

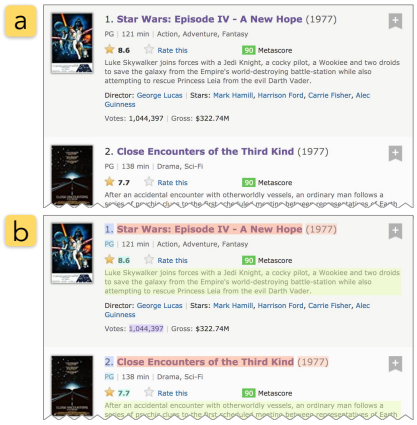


Figure 5. A centralized server stores relations used in prior Rousillon programs; Rousillon uses stored relations to communicate about loops it can add. When a user hovers over a known relation during demonstration, Rousillon highlights the relation to indicate it knows how to loop over it. Here, (a) shows an IMDb.com page, 1977's top box office movies, and (b) shows the same page with the movies relation highlighted.

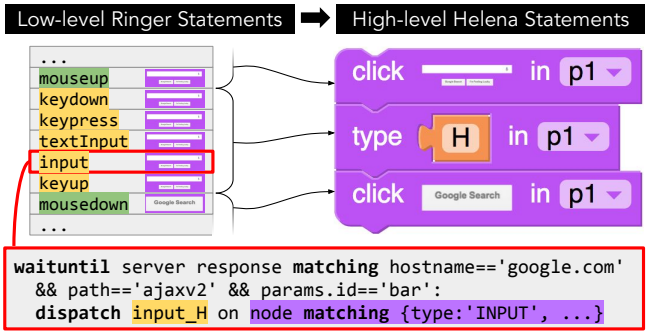
Gulf of Evaluation. When the user scrapes a new element, Rousillon adds its content to a preview of the first row, displayed in the control pane (see Fig. 4). This preview of the output data reduces the gulf of evaluation by giving the user information about the program Rousillon is building. Rousillon's output program will always produce the previewed row of data as its first output row, as long as the structure and content of the webpages remain stable. If at any point the user realizes that the preview does not show the intended data, they can identify without running the program – without even finishing the demonstration – that the current demonstration will not produce the desired program.

Finishing. When the user ends a demonstration by clicking the 'Stop Recording' button (see Fig. 2(e)), the recorded interaction is passed as input to Rousillon's PBD system, described in the Algorithms section. The output of the PBD system is a program in the high-level Helena web automation language; the control pane displays this program in a Scratch-style [45] blocks-based editor (see output in Fig. 3). At this stage, the user can edit, run, and save the new program.

ALGORITHMS

In this section, we describe three key Rousillon components: the Reverse Compiler, Relation Selector, and Generalizer. Fig. 3 shows how these components interact to write a scraping program based on an input demonstration.

Reverse Compiler



Problem Statement: Traditional replayers write high-level, readable scripts but are fragile on today's interactive, AJAX-heavy webpages; modern replayers work on interactive pages but write low-level, unreadable programs.

Our Solution: Use a modern replayer for robustness, but reverse-compile to the Helena language to recover readability.

Rousillon uses Ringer to record user demonstrations (Recorder in Fig. 3). We chose Ringer because it outperforms alternatives on robustness to page content and design changes [4]. However, the output of a Ringer recording is a script in Ringer's low-level language, with statements like the red-outlined statement in the Reverse Compiler figure. With one statement for each DOM event – e.g., `keydown`, `mouseup` – a typical Ringer script has hundreds of statements, each tracking more than 300 DOM node attributes. This exhaustive tracking of low-level details makes Ringer scripts robust, but it also makes them unreadable even for DOM experts. To meet Design Goal D1, sheltering users from browser internals, we hide these details.

The Reverse Compiler translates Ringer programs into a high-level web automation language, Helena. It produces high-level statements that a user can read, but whose semantics are determined by associated snippets of code in the Ringer language – the low-level statements that make replay robust.

The Reverse Compiler slices the input Ringer program into sequences of consecutive statements that operate on the same target DOM node and are part of one high-level action; then it maps each sequence of Ringer statements to a single Helena statement. Each Ringer statement includes the type, t, of its DOM event and the DOM node, n, on which it dispatches the event. Browsers use a fixed set of DOM event types, T, and Helena has a fixed set of statements, H. Rousillon uses a map $m : T \mapsto H$ that associates each type in T with a high-level Helena statement; for example, `keydown`, `keypress`, `textInput`, `input`, and `keyup` all map to Helena's `type` statement. Rousillon uses m to label each Ringer statement: $(t,n) \rightarrow (t,n,m(t))$. Next it slices the Ringer program into sequences of consecutive statements that share the same n and $m(t)$[1]. Because many event types are mapped to a single Helena statement type, the Ringer sequences are typically long. Next, the Reverse Compiler writes the Helena program; it maps each sequence of Ringer statements to a single Helena statement that summarizes the effects of the whole Ringer sequence. The Reverse Compiler figure illustrates this process; note that a slice of typing-related Ringer statements are mapped to a single Helena statement: `type "H" in p1`.

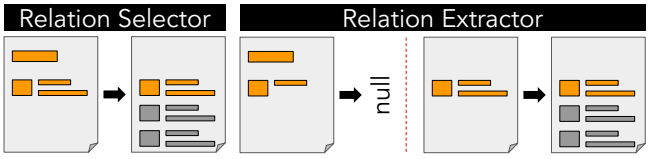
[1]This is a slight simplification. Statements for DOM events that are invisible to human users (e.g., `focus`), can have different n.

```
curSize = |N|
while curSize > 0 do
    S = subsetsOfSize(N, curSize)
    for subset in S do
        rel = relationExtractor(w, subset)
        if rel then
            return rel
    end
    curSize = curSize - 1
end
```

Algorithm 1. Relation Selector algorithm. Input: w, a webpage, and N, a set of interacted nodes in w. Output: rel, a relation of nodes in w that maximizes: $|\{n : n \in N \wedge n \in rel[0]\}|$

Each Helena statement stores its Ringer sequence, which the Helena interpreter runs to execute the statement. Thus the Reverse Compiler's output is a loop-free Helena program that runs the same underlying Ringer program it received as input.

Relation Selector



Problem Statement: To support a single-demonstration interaction model, we must determine for each recorded action (i) whether the action should be repeated on additional webpage elements and (ii) if yes, which additional webpage elements. In short, we must find relations relevant to the task.

Our Solution: A relation selection algorithm to extract information about relations from a straight-line interaction script.

The central goal of our Relation Selector is to predict, based on a single loop-free replay script, whether the script interacts with any relations over which the user may want to iterate. Our Relation Selector approach has two key components:

- **Relation Selector:** The top-level algorithm. It takes a replay script as input and produces a set of relations as output.
- **Relation Extractor:** A subroutine. It takes a set of nodes as input. If it can find a relation that has one column for each input node, it returns the relation; if not, it returns null.

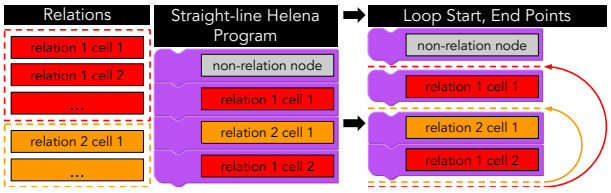
Relation Selector. The Relation Selector uses a feedback loop with the relation extractor. It starts by identifying the set of all DOM nodes with which the input replay script interacts, then groups them according to the webpages on which they appear. For each webpage w and its set of nodes N, the Relation Selector applies Algorithm 1. Essentially the Relation Selector repeatedly calls the relation extractor on subsets of N, from the largest subsets to the smallest, until it finds a subset of nodes for which the relation extractor can produce a relation. For instance, if a user interacted with a movie title, movie rating, and webpage title on a single webpage, the Relation Selector would first seek a relation that includes all three of those nodes in the first row. If the relation extractor fails to find a relation (the likeliest outcome given standard web design patterns), the Relation Selector tries subsets – e.g., subsets of size two until it finds a well-structured relation that includes the movie title and movie rating (but not the page title) in the first row.

Relation Extractor. Our custom relation extractor is closely related to the relation extractors [53, 29, 22, 20] discussed in Related Work, with one key difference: it is designed to excel in the case of having only one row of data. We found that prior relation extractor techniques often required at least two rows of data as input. Since being able to suggest a good relation given only a single row of data is critical to our interaction

model, we developed a custom relation extractor adapted to this constraint. The key insight is to fingerprint the structure of the input cells' deepest common ancestor (DCA), then find a sibling of the DCA that shares the structure fingerprint (see illustration in Appendix A). For instance, if the path through the DOM tree from the DCA n to an actor name is $p1$ and the path from n to the actor's role is $p2$, the relation extractor would seek a sibling of n that also has descendant nodes at the $p1$ and $p2$ positions. Using the sibling node as a second row of labeled cells, we can apply the same techniques that drive prior relation extractors.

Saved Relations. A centralized server stores a database of relation extractors used in past Rousillon programs. When the Relation Selector chooses which relation to extract from a page, it considers both freshly identified relations and stored relations that originated on a related page and function on the current page. It ranks them, preferring relations that: include as many of the input nodes as possible, have many rows on the current page, and have been used in prior scripts. With this approach, users can expect that Rousillon already knows how to find relations on many mainstream sites.

Generalizer



Problem Statement: Scraping hierarchical data requires nested loops, with one loop for each level of the hierarchy.

Our Solution: A Generalizer that introduces nested loops.

The Generalizer takes a loop-free Helena program and a set of relations as input. It has three roles: (i) adapt the program to store output data, (ii) for each relation, determine where to insert a loop over the relation, and (iii) parameterize the loop bodies to work on loop variables rather than concrete values.

Output Statement. The input program collects data from DOM nodes but does not store the data or accumulate any output. The Generalizer appends an `output` statement at the tail of the Helena program to add a row of data to an output dataset; it adds a cell in the `output` statement for each `scrape` statement in the program. Thus a program that scrapes a movie title and an actor name produces [movie, actor] rows. If the actor is scraped in a loop, the program produces multiple rows with the same movie but different actors. This builds a 'universal relation' view of the hierarchical data, a join over all relations. This meets our data science collaborators' requests for tabular data that they can manipulate with familiar spreadsheet and statistical software.

Loop Insertion. The input program is loop-free, but the output program should iterate over all relations identified by the Relation Selector. To add loops, the Generalizer first identifies whether each Helena statement's target node appears in any relations. To identify the hierarchical structure, we produce a total order over relations based on the order in which their nodes appear in the program. The first relation whose cells are used is highest in the hierarchy (associated with the outermost loop); the relation that sees its first cell use only after all other relations have been used is lowest in the hierarchy (associated with the innermost loop). For each relation in the hierarchy, we insert a loop that starts immediately before the first use of the relation's cells and ends after the output statement.

Parameterization. The input program operates on cells of the first dataset row, so after loop insertion, each loop body still only operates on the first row. For example, if the goal is to scrape 100 movies, the program at this intermediate point scrapes the *first* movie 100 times. The Generalizer adapts loop bodies so that we can apply them to multiple rows of a relation. We use *parameterization-by-value*, a metaprogramming technique for turning a term that operates on a concrete value into a function that can be called on other values. Essentially, $(pbv\ term\ value) \rightarrow (lambda(x)\ term')$, where $term'$ is $term$ with all uses of $value$ replaced with uses of a fresh variable x. We execute parameterization-by-value for DOM nodes, typed strings, and URLs. For each Helena statement in a newly inserted loop, we check whether the target node appears in the loop's associated relation. If yes, we identify the index i of the target node n in the relation row. We then replace the Helena statement's slice of Ringer events E with $(pbv\ E\ n)(row[i])$, where *row* is the variable name used to refer to the relation row at each iteration. We repeat this process for typed strings, for each `type` statement in a loop (checking whether the typed string includes the text of any relation node), and for URLs, for each `load` statement in the loop.

Range

Data Shape. The depth of a Rousillon program's deepest nested loop is the bound on the depth of the collected hierarchy; our Generalizer can add any number of nested loops, so Rousillon can collect arbitrarily deep hierarchies. Sibling subtrees can have different depths – e.g., a movie can have no actors and thus no child nodes. The number of child nodes of any given node is unbounded and can vary – e.g., one movie can have 10 actors while another has 20. Thus, the number of dataset rows is unbounded. The number of columns per row is bounded by the number of scrape statements in the program; recall the Generalizer adds an output statement that adds rows with one cell for each scrape statement – e.g., if the user scraped a movie title and an actor name, each output row includes a movie title and an actor name. To collect variable-length data, users should add additional layers in the data hierarchy rather than variable-length rows – e.g., if users want a list of actors for each movie, they collect many [movie, actor] rows rather than one variable-length [movie, actor1, actor2,...] row per movie. The number of scrape statements bounds but does not precisely determine the number of populated cells per row; Rousillon leaves an empty cell for missing data – e.g., if an actor appears in a cast list without an associated character name, the role cell is left empty.

Node Addressing. "Node addressing" or "data description" is the problem of identifying the node on which to dispatch an action or from which to extract data. This is a long-standing problem in the web PBD community and largely determines how well a program handles page redesigns and other changes. Rousillon uses Ringer for replay and thus uses Ringer's node addressing (see [4]). This means any webpage changes that Ringer cannot handle, Rousillon also cannot handle.

Ambiguity. Rousillon uses a single demonstration as input to the PBD process, so inputs can be ambiguous. For instance, say a user scrapes a table in which some rows are user-generated posts and some rows are ads. The user may

want to scrape all rows or only rows that have the same type as the demonstrated row. A single-row demonstration is insufficient to distinguish between these two cases. Thus it is critical that users have the option to edit output programs; this motivated our use of the high-level Helena language and a blocks-based editor (see output in Fig. 3). Although this paper focuses on the learnability of Rousillon's PBD interaction, studying the editability of its output programs is a natural next step and critical to understanding whether Rousillon's ambiguity-embracing approach is practical for real users.

USER STUDY

To evaluate Rousillon, we conducted a within-subject user study comparing Rousillon with Selenium, a traditional web automation library. Based on Design Goal D3, we were more interested in the learnability than the usability of the tool, so we focused our study on the following research questions:

- RQ1: Can first-time users successfully learn and use Rousillon to scrape distributed hierarchical data?
- RQ2: Will it be easier to learn to complete this task with Rousillon or with a traditional web automation language?

We recruited 15 computer science graduate students (9 male, 6 female, ages 20 to 38) for a two-hour user study. All participants had been programmers for at least 4 years and had used Python (one of Selenium's host languages).

Procedure

We started each session with a description of the participant's assigned web scraping task. Each participant was assigned one of two tasks: (i) Authors-Papers: Starting at Google Scholar, iterate through a list of authors, and for each author a list of papers. The authors list is a multi-page list; each author links to a profile page that lists papers; more papers can be loaded via a 'More' button that triggers AJAX requests. (ii) Foundations-Tweets: Starting at a blog post that lists charitable foundations, iterate through charitable foundations, and for each foundation a list of tweets. The foundations list is a single-page list; each foundation links to a Twitter profile that lists tweets; more tweets can be loaded by scrolling the page to trigger AJAX requests. Authors-Papers came from [8], and Foundations-Tweets from a Public Policy collaborator.

Next, we asked each participant to complete the assigned task with two tools: Rousillon and Selenium [49] (in particular, the Python Selenium library). The tool order was randomized to distribute the impact of transferable knowledge. For each tool, we pointed the user to a webpage with documentation of the tool but left the choice about whether and how to use that documentation (or any other resources available on the web) up to the participant; being primarily interested in learnability, we wanted to observe the time between starting to learn a tool independently and understanding it well enough to complete a task with it, so we let participants control the learning experience. If a participant did not complete the task with a given tool within an hour, we moved on to the next stage.

After using both tools, each participant answered a survey that asked them to reflect on the tools' learnability and usability.

Comparison Against Traditional Programming. By conducting a within-subject user study, we obtain a fair evaluation of the productivity benefits of PBD scraping versus traditional scraping for realistic datasets. However, this design choice limits our participants to coders – novices with Rousillon, but not novice programmers. We leave evaluation of Rousillon's usability by end users for future work. Note that a comparison against state-of-the-art PBD web automation is not possible, since no existing PBD tool can scrape the distributed hierarchical datasets we use in our tasks.

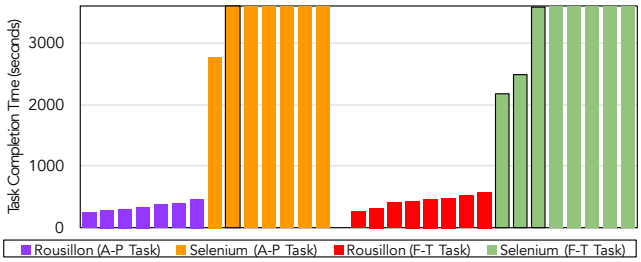


Figure 6. Combined learning and task completion times for the Authors-Papers and Foundations-Tweets tasks, using Rousillon and Selenium. The cutoff was one hour; Selenium bars that extend to the one hour mark indicate the participant was unable to complete the task with Selenium. All participants succeeded with Rousillon in under 10 minutes. Only four of 15 succeeded with Selenium. Four Selenium bars are marked with a black outline, indicating the participant had prior experience with Selenium. No participant had prior experience with Rousillon.

Results

User Performance

Fig. 6 shows completion times. Recall that since we are most interested in learnability, which we operationalized as time between starting to learn a tool and producing a first correct program with it, we did not distinguish between time spent learning and time spent writing programs. Thus, completion times include both learning and program authoring time.

All participants completed their tasks with Rousillon, for a completion rate of 100%. In contrast, only four out of 15 participants completed their tasks with Selenium, for a completion rate of 26.7%. Rousillon was also fast. All participants learned and successfully used Rousillon in under 10 minutes, four in less than 5 minutes. The median time was 6.67 minutes.

For the Authors-Papers task, the average completion time with Rousillon was 5.7 minutes. With timed-out participants' times truncated at 60 minutes, the average completion time with Selenium was 58.0 minutes. For the Foundations-Tweets task, the average completion time with Rousillon was 7.3 minutes. With timed-out participants' times truncated at 60 minutes, the average completion time with Selenium was 54.7 minutes. Because our data is right-censored, with all Selenium times above 60 minutes known only to be above 60 minutes, we cannot provide a precise estimate of the learnability gains for Rousillon versus Selenium. However, we can offer a lower bound: the time to learn Selenium well enough to complete our tasks is at least 8.5x higher than the time to learn Rousillon.

Although excluding right-censored data leaves only four data points, Rousillon's effect on performance is statistically significant even for this small sample. We conducted a paired-samples t-test for participants who completed the task with both tools. There was a significant difference in the completion times for the Selenium (M=2751.2, SD=600.2) and Rousillon (M=405.8, SD=173.1) conditions; $t(3)=8.534$, $p = 0.0034$.

To evaluate timed out participants' progress towards a working Selenium scraper, we defined five checkpoints on the way

to a complete scraper. Completing all checkpoints means completing the task. We found that six of 15 participants never reached any of the checkpoints with Selenium. See Appendix B for a table of which checkpoints each participant reached.

We also measured the number of debugging runs as an additional measure of work. On average, when participants used Rousillon, they ran 0.2 debugging runs. (Of 15 participants, 13 only ran a correct program and thus had no debugging runs.) On average, using Selenium, they ran 20.8 debugging runs. Again, Selenium values are right-censored, since participants who did not finish might need more debugging runs.

Although all participants were first-time Rousillon users, 4 of 15 participants had used Selenium in the past. Those with prior Selenium experience accounted for 3 of the 4 participants who completed their tasks with Selenium; one participant with prior Selenium experience was unable to complete the task, and only one participant with no prior Selenium experience was able to complete a task with Selenium.

User Perceptions

We were interested in whether our participants, being programmers, would prefer a PBD or a traditional programming approach. In the survey after the programming tasks, we asked participants to rate how usable and learnable they found the tools. Participants answered all questions with a seven-point Likert scale, with 1 indicating the tool was very easy to use or learn and 7 indicating the tool was very difficult to use or learn. The average ratings for Rousillon and Selenium *usability* were 1.2 and 4.8, respectively. The average ratings for Rousillon and Selenium *learnability* were 1.1 and 5.6, respectively.

We also asked what tools participants would want for future scraping tasks of their own. Participants could list as many tools as they liked, including tools they knew from elsewhere. All but one participant (93.3%) indicated they would use Rousillon; two (12.5%) indicated they would use Selenium.

User Study Discussion

What is challenging about traditional web automation?

We asked participants what was hardest about using Selenium. Answers ranged from "Everything" to "Selecting elements on a webpage designed by someone else; creating new windows or tabs (supposedly possible...); expanding out paginated data tables." to "Mystifying browser behaviors (are we waiting for a page to load? is an element visible? is the element findable but not clickable????)", "I worry that my Selenium script is not very robust: I might have missed a lot of cases or used ambiguous selectors to find elements." The common threads across responses were: (i) interacting with low-level representations, (ii) understanding target pages' structures and behaviors, (iii) finding appropriate library methods, and (iv) concerns about robustness to webpage structure changes.

What is challenging about PBD web automation?

In contrast, the most common answer to what was hardest about using Rousillon was variations of "Nothing" or "NA"; nine of 15 participants provided these answers. The primary concerns were about control: "to what extent would people be able to tweak things outside of the automatic framework?", "I think if I got used to using Selenium regularly, I would feel limited by Rousillon." This impression may be the effect of our study design, which did not require participants to, for instance, add if statements or other custom control flow.

What is good about traditional web automation?

Participants perceived Selenium as offering more low-level control: "More manual work so maybe more flexibility (?)", "Very fine grained control if you have the time to figure it all out." Others suggested Selenium might be more useful in the context of broader programming tasks: "The resulting script could be added as part of a larger software pipeline" or "you could run it headless and in parallel on multiple computers." This suggests it is easier for programmers to imagine programmatically manipulating a Selenium program than a Rousillon program, although the suggested manipulations – using a scraper in a larger pipeline or running in a distributed setting – are possible with Rousillon and indeed already practiced by real users.

What is good about PBD web automation?

Participants appreciated that Rousillon handled the hierarchical structure for them ("[It] was very useful how it automatically inferred the nesting that I wanted when going to multiple pages so that I didn't have to write multiple loops.") and how it shielded them from low-level node finding and relation finding ("Locating the thing I care about is much simpler: just click on it"). They also felt it anticipated their needs: "I didn't know anything about web scraping before starting and it fulfills a lot of the functionality I didn't even expect I would need," "Super easy to use and I trust that it automatically follows the links down. It felt like magic and for quick data collection tasks online I'd love to use it in the future."

How do traditional and PBD web automation compare?

We also solicited open-ended responses comparing Rousillon and Selenium, which produced a range of responses: "The task was an order of magnitude faster in Rousillon," "Rousillon is much, much quicker to get started with," "If I had to use Selenium for web scraping, I would just not do it," "Rousillon's way easier to use – point and click at what I wanted and it 'just worked' like magic. Selenium is more fully featured, but...pretty clumsy (inserting random sleeps into the script)," "Rousillon is a self balancing unicycle, Selenium is a 6ft tall unicycle. One you can buy/download and [you're] basically up and going. The other you needs years of practice just to get to the place where you feel comfortable going 10 ft." We were interested in what comparisons participants chose to make. Ultimately, the comparisons sorted from highest to lowest frequency were about: programming time, ease-of-use, ease-of-learning, language power, and program robustness.

CONCLUSION AND FUTURE WORK

Rousillon's novel interaction model and generalization algorithms push PBD web scraping beyond the restrictions that prevented past tools from collecting realistic datasets – in particular, distributed, hierarchical data. With a learning process at least 8x faster than the learning process for traditional scraping, this strategy has the potential to put web automation tools in the hands of a wider and more diverse audience. We feel this motivates two key future directions: (i) Although Rousillon is designed to combine PBD with a learnable, editable programming language, this paper only evaluates PBD. Evaluating Helena's editability is a critical next stage. (ii) Rousillon was designed with social scientists – and end users generally – in mind. Because this paper compares PBD to traditional programming, our evaluation focused on a population that can use traditional languages. We see testing with end users as a crucial next step on the path to democratizing web data access.

REFERENCES

1. a9t9. 2018. Web Browser Automation with KantuX, Desktop Automation, OCR - Fresh 2017 Robotic Process Automation (RPA). (March 2018). https://a9t9.com/
2. Brad Adelberg. 1998. NoDoSE - A tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents. In *SIGMOD Record*.
3. Nikolaos Askitas and Klaus F. Zimmermann. 2015. The internet as a data source for advancement in social sciences. *International Journal of Manpower* 36, 1 (2015), 2–12. DOI: http://dx.doi.org/10.1108/IJM-02-2015-0029
4. Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 748–764. DOI:http://dx.doi.org/10.1145/2983990.2984020
5. Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. 2006. A Survey of Web Information Extraction Systems. *IEEE Trans. on Knowl. and Data Eng.* 18, 10 (Oct. 2006), 1411–1428. DOI: http://dx.doi.org/10.1109/TKDE.2006.152
6. Kerry Shih-Ping Chang and Brad A. Myers. 2017. Gneiss. *J. Vis. Lang. Comput.* 39, C (April 2017), 41–50. DOI: http://dx.doi.org/10.1016/j.jvlc.2016.07.004
7. Sarah Chasins. 2017. Helena: Web Automation for End Users. http://helena-lang.org/. (Oct. 2017).
8. Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. 2015. Browser Record and Replay As a Building Block for End-User Web Automation Tools. In *Proceedings of the 24th International Conference on World Wide Web Companion (WWW '15 Companion)*. 179–182. DOI: http://dx.doi.org/10.1145/2740908.2742849
9. Sarah Chasins and Rastislav Bodik. 2017. Skip Blocks: Reusing Execution History to Accelerate Web Scripts. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 51 (Oct. 2017), 28 pages. DOI:http://dx.doi.org/10.1145/3133875
10. Allen Cypher. 1991. EAGER: Programming Repetitive Tasks by Example. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. ACM, New York, NY, USA, 33–39. DOI: http://dx.doi.org/10.1145/108844.108850
11. Sergio Flesca, Giuseppe Manco, Elio Masciari, Eugenio Rende, and Andrea Tagarelli. 2004. Web Wrapper Induction: A Brief Survey. *AI Commun.* 17, 2 (April 2004), 57–61. http://dl.acm.org/citation.cfm?id=1218702.1218707
12. Tim Furche, Jinsong Guo, Sebastian Maneth, and Christian Schallhart. 2016. Robust and Noise Resistant Wrapper Induction. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 773–784. DOI:http://dx.doi.org/10.1145/2882903.2915214
13. Greasemonkey. 2015. Greasemonkey :: Add-ons for Firefox. https://addons.mozilla.org/en-us/firefox/addon/greasemonkey/. (Nov. 2015).
14. Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. (January 2011).
15. HaskellWiki. 2017. HXT - HaskellWiki. https://wiki.haskell.org/HXT#Selecting_text_from_an_HTML_document. (Nov. 2017).
16. hpricot. 2015. hpricot/hpricot. https://github.com/hpricot/hpricot. (Aug. 2015).
17. Chun-Nan Hsu and Ming-Tzung Dung. 1998. Generating finite-state transducers for semi-structured data extraction from the Web. *Information Systems* 23, 8 (1998), 521 – 538. DOI: http://dx.doi.org/10.1016/S0306-4379(98)00027-1 Semistructured Data.
18. Darris Hupp and Robert C. Miller. 2007. Smart bookmarks: automatic retroactive macro recording on the web. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. ACM, New York, NY, USA, 81–90. DOI: http://dx.doi.org/10.1145/1294211.1294226
19. David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling web browsers to augment web sites' filtering and sorting functionalities. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*. ACM Press, New York, NY, USA, 125–134. DOI: http://dx.doi.org/10.1145/1166253.1166274
20. Import.io. 2016. Import.io | Web Data Platform & Free Web Scraping Tool. (March 2016). https://www.import.io/
21. iOpus. 2013. Browser Scripting, Data Extraction and Web Testing by iMacros. http://www.iopus.com/imacros/. (July 2013).
22. KimonoLabs. 2016. Kimono: Turn websites into structured APIs from your browser in seconds. (March 2016). https://www.kimonolabs.com
23. Andhy Koesnandar, Sebastian Elbaum, Gregg Rothermel, Lorin Hochstein, Christopher Scaffidi, and Kathryn T. Stolee. 2008. Using Assertions to Help End-user Programmers Create Dependable Web Macros. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 124–134. DOI: http://dx.doi.org/10.1145/1453101.1453119
24. Nicholas Kushmerick. 1997. *Wrapper Induction for Information Extraction*. Ph.D. Dissertation. AAI9819266.
25. Nicholas Kushmerick. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence* 118, 1 (2000), 15 – 68. DOI: http://dx.doi.org/10.1016/S0004-3702(99)00100-9

26. Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. 1997. Wrapper Induction for Information Extraction. In *Proc. IJCAI-97*. http://citeseer.nj.nec.com/kushmerick97wrapper.html

27. Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1-2 (Oct. 2003), 111–156. DOI: http://dx.doi.org/10.1023/A:1025671410623

28. Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 527–534. http://portal.acm.org/citation.cfm?id=657973

29. Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553. DOI: http://dx.doi.org/10.1145/2594291.2594333

30. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1719–1728. DOI: http://dx.doi.org/10.1145/1357054.1357323

31. Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 723–732. DOI:http://dx.doi.org/10.1145/1753326.1753432

32. James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user programming of mashups with Vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

33. Jalal Mahmud and Tessa Lau. 2010. Lowering the barriers to website testing with CoTester. In *Proceedings of the 15th international conference on Intelligent user interfaces (IUI '10)*. ACM, New York, NY, USA, 169–178. DOI: http://dx.doi.org/10.1145/1719970.1719994

34. Noortje Marres and Esther Weltevrede. 2013. Scraping the Social? *Journal of Cultural Economy* 6, 3 (2013), 313–335. DOI: http://dx.doi.org/10.1080/17530350.2013.772070

35. Mozenda. 2018. Web Scraping Solutions for Every Need. (March 2018). https://www.mozenda.com/?utm_source=googleadwords&utm_medium=cpc&utm_term=Mozenda&gclid=EAIaIQobChMIuM71jfGc2gIVC7nACh2mOwz8EAAYASAAEgLSzPD_BwE

36. Ion Muslea, Steve Minton, and Craig Knoblock. 1999a. A Hierarchical Approach to Wrapper Induction. In *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS '99)*. ACM, New York, NY, USA, 190–197. DOI: http://dx.doi.org/10.1145/301136.301191

37. Ion Muslea, Steve Minton, and Craig Knoblock. 1999b. A Hierarchical Approach to Wrapper Induction. In *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS '99)*. ACM, New York, NY, USA, 190–197. DOI: http://dx.doi.org/10.1145/301136.301191

38. Nokogiri. 2016. Tutorials - Nokogiri. http://www.nokogiri.org/. (Nov. 2016).

39. Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books, Inc., New York, NY, USA.

40. Octoparse. 2018. Web Scraping Tool & Free Web Crawlers for Data Extraction | Octoparse. (March 2018). https://www.octoparse.com/

41. SIMILE Semantic Interoperability of Metadata and Information in unLike Environments. 2016. Solvent. (March 2016). http://simile.mit.edu/solvent/

42. Adi Omari, Sharon Shoham, and Eran Yahav. 2017. Synthesis of Forgiving Data Extractors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, NY, USA, 385–394. DOI: http://dx.doi.org/10.1145/3018661.3018740

43. ParseHub. 2018. Free web scraping - Download the most powerful web scraper | ParseHub. (March 2018). https://www.parsehub.com/

44. Platypus. 2013. Platypus. http://platypus.mozdev.org/. (Nov. 2013).

45. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. DOI:http://dx.doi.org/10.1145/1592761.1592779

46. Leonard Richardson. 2016. Beautiful Soup: We called him Tortoise because he taught us. http://www.crummy.com/software/BeautifulSoup/. (March 2016).

47. Scrapinghub. 2018. Visual scraping with Portia. (March 2018). https://scrapinghub.com/portia

48. Scrapy. 2013. Scrapy. http://scrapy.org/. (July 2013).

49. Selenium. 2013. Selenium-Web Browser Automation. http://seleniumhq.org/. (July 2013).

50. Selenium. 2016. Selenium IDE Plugins. http://www.seleniumhq.org/projects/ide/. (March 2016). http://www.seleniumhq.org/projects/ide/

51. Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

52. Howard T Welser, Marc Smith, Danyel Fisher, and Eric Gleave. 2008. Distilling digital traces: Computational social science approaches to studying the internet. *Handbook of online research methods* (2008), 116–140.

53. Jeffrey Wong and Jason I. Hong. Making Mashups with Marmite: Towards End-user Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Syst24.1240842*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/1240624.1240842

54. Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and C. Lee Giles. 2009. Efficient Record-level Wrapper Induction. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM '09)*. ACM, New York, NY, USA, 47–56. DOI: http://dx.doi.org/10.1145/1645953.1645962

APPENDIX

RELATION SELECTOR EXAMPLE

Here we expand on the sample Relation Selector execution described in the Algorithms section. Say the user interacts with a webpage with the partial DOM tree depicted in Fig. 7 and scrapes the nodes highlighted in red: `page title`, `movie 1 title`, and `movie 1 rating`.

The Relation Selector starts by passing all three interacted nodes to the relation extractor, which tries to find a relation with all three nodes in the first row. The deepest common ancestor of all three is `page body`. The relation extractor identifies a structure fingerprint for `page body`; the fingerprint is a representation of the Fig. 7 edges highlighted in orange and purple – that is, the paths to all three interacted nodes. Next it looks for a sibling of `page body` that has nodes at all of those positions. Since `page header` does not have nodes at those positions, the relation extractor fails to find an appropriate sibling node (representing a predicted second row) for this set of input nodes. The relation extractor returns null for this input.

Next the Relation Selector tries subsets of the three interacted nodes, starting with subsets of size two. Eventually it tries { `movie 1 title, movie 1 rating` }. For this subset, the deepest common ancestor is `movie 1`; its structure fingerprint includes the paths highlighted in purple. The relation extractor seeks a sibling of `movie 1` that has nodes at those paths and finds that `movie 2` has nodes `movie 2 title` and `movie 2 rating` at those positions. Since this subset produces a first row that has an appropriate sibling for making a second row, the relation extractor passes the first and second rows to a traditional relation extraction algorithm ([53, 29, 22, 20]), and the Relation Selector returns the result.

For simplicity, we do not describe cases in which Rousillon extracts multiple relations from one page, but this is necessary in practice.

TASK COMPLETION LEVELS WITH SELENIUM

Task	1 full row	1st pg outer loop	2nd pg outer loop	1st pg inner loop	2nd pg inner loop
A-P					
A-P					
A-P					
A-P	✓				
A-P	✓	✓		✓	
A-P	✓	✓		✓	
A-P	✓	✓	✓	✓	✓
F-T			-		
F-T			-		
F-T			-		
F-T		✓	-		
F-T		✓	-		
F-T	✓	✓	-	✓	✓
F-T	✓	✓	-	✓	✓
F-T	✓	✓	-	✓	✓

Table 1. Because we wanted to understand partial completions of tasks with Selenium, we defined subtasks of the Authors-Papers and Foundations-Tweets tasks. We imposed higher standards for Rousillon programs, but for the Selenium component of the study, we considered a task complete once the participant wrote a script that completed all five of these subtasks. The subtasks are: (i) producing a row with all requisite cells, (ii) iterating through the first page of the relation associated with the outer loop, (iii) iterating through at least the first two pages of the relation associated with the outer loop, (iv) iterating through the first page of the relation associated with the inner loop, and (v) iterating through at least the first two pages of the relation associated with the inner loop. Note that the Foundations-Tweets task has only four subtasks because the list of foundations appears entirely in a single webpage; we show '-' in the '2nd pg outer loop' column to indicate there is no second page of data for this loop. Overall, less than half of the participants wrote a script that could collect one complete row of the target dataset within the one-hour time limit.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and our shepherd, Philip Guo, for their feedback. This work is supported in part by NSF Grants CCF–1139138, CCF–1337415, NSF ACI–1535191, and NSF 16-606, a Grant from U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers program under Award Number FOA–0000619, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

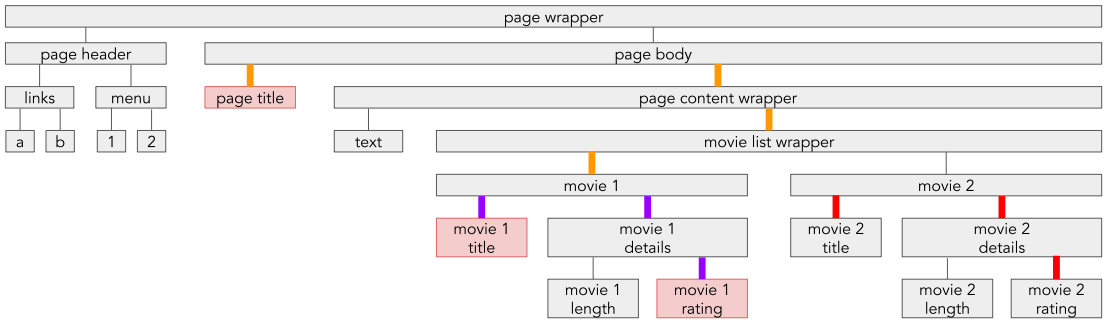


Figure 7. A sample DOM tree. If the user interacts with the nodes highlighted in red, the Relation Selector will produce a relation with one column for movie titles and one column for movie ratings. The relation extractor will find that `movie 1`'s structure fingerprint (the paths highlighted in purple) matches the structure fingerprint of `movie 2` (the paths highlighted in red).