Improving Web Automation Tools through UI Context and Demonstration

by

Rebecca P. Krosnick

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Rebecca P. Krosnick

rkros@umich.edu

ORCID iD: 0000-0002-8427-3895

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of a number of people in my life. I would like to thank:

- First and foremost, my advisor, Steve Oney. Beyond his guidance on research (which has taught me so much about end-user programming, choosing meaningful research problems, scoping projects, and rapid prototyping), what I have appreciated the most about working with Steve is his kindness. The PhD was challenging for me at many times and I appreciate Steve's patience and encouragement during those times. I appreciate how he listened to my ideas and questions with curiosity and without judgment, making it a safe space for me to explore and make mistakes. Finally, I appreciate how Steve cared about me and his other students as people beyond just research. His support has meant so much to me. Thank you, Steve.

- The rest of my committee, for their time and feedback and for helping me think about the broader implications of my work: Elena Glassman, Anhong Guo, Colleen Seifert, Xinyu Wang.

- My internship mentors and collaborators, who have broadened my horizons, given me fun summer experiences, and helped shape my PhD experience: Fraser Anderson, Justin Matejka, George Fitzmaurice, Tovi Grossman, Bongshin Lee, Ken Hinckley, Michel Pahud, Nathalie Riche, John Thompson, Hugo Romat, Amanda Swearngin, Jeffrey Nichols, Jason Wu, Eldon Schoop, Jeffrey P. Bigham.

- Other collaborators, especially Sang Won Lee and Fei Wu.

- UMich computer science and HCI faculty, who have been friendly faces around the department; taken an interest in me, my work, and my success; and given me advice and feedback. Thanks especially to Anhong Guo, Xinyu Wang, Mark Guzdial, Nikola Banovic, Alanson Sample, Xu Wang, Michael Nebeling, Cyrus Omar, Wes Weimer, Tawanna Dillahunt.

- Other researchers in the HCI and VL/HCC communities who have supported me and my work over the years.

- The National Science Foundation for supporting my work for the past few years under NSF Award 2007857.
- All the friends I've made during the PhD who have brought great fun to my experience and whose support has meant so much: Divya Ramesh, Vaishnav Kameswaran, Amani Alkayyali, Jaylin Herskovitz, Katie Cunningham, Eshwar Chandrasekharan, Lei Zhang, April Wang, Ashley Zhang, Mauli Pandey, Jiacheng Zhang, Anthony Liu, John Chung, Yiwei Yang, Zhefan Ye, Harman Kaur, Sai R. Gouravajhala, Penny Trieu, Jane Im, Jordan Huffaker, Preeti Ramaraj, Tara Safavi, Xiaoying Pu, Nel Escher, Snehal Prabhudesai, Sumit Asthana, Anindya Das Antar, Anjali Singh, Yasha Iravantchi, Madeline Endres, Andrew Blinn, David Moon, Ihudiya Finda Williams, Yu Huang, Agrima Seth, Hari Subramonyam, Nischal Shrestha, Nikhita Joshi, Cheryl Lao, Roya Shams, Kimia Kiani, Seonghyeon Moon, Molly Jane Nicholas, Nimo Ni, Gabriel Matute, Luis Morales-Navarro, Nava Haghighi, Venkatesh Potluri, Sam Robertson, Jay Wang, Parsa Nooralinejad.
- The Palter, Lavine, Geller, and Krosnick families and other extended family for all their love and support over the years.
- High school, college, work, and family friends, especially Kalyani Kumar.
- My mom and sister, who have been there for me every step of the way, every day, through all the ups and downs professionally and personally. I could not have done this without them.
- My dad and my aunt Susan who are no longer with us but were a huge part of my life and ardent supporters of my education growing up, and my dad especially for encouraging my interest in STEM which has led me to where I am today.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

UI User Interface

PBD Programming by Demonstration

PBE Programming by Example

DOM Document Object Model

API Application Programming Interface

IDE Integrated development environment

HTML HyperText Markup Language

CSS Cascading Style Sheets

XPath XML Path Language

AI Artificial Intelligence

ML Machine Learning

NL Natural Language

NLU Natural Language Understanding

NLP Natural Language Processing

LLM Large Language Model

ABSTRACT

User interface (UI) automation allows people to perform UI tasks programmatically and can be helpful for computer or smartphone tasks that are tedious, repetitive, or inaccessible. UI automation works by programmatically mimicking a user's interactions on a UI, for example clicking a button or typing into a text field. Traditionally people create UI automation macros by writing code, which requires programming expertise and familiarity with UI technologies. Researchers have explored direct manipulation interfaces and programming-by-demonstration (PBD) to make creating UI automation more accessible for people with less programming experience. With PBD, the user provides demonstrations of how they want their program to behave in a small set of scenarios, and the system then infers a generalized program. Since demonstrations are inherently ambiguous, a key challenge of PBD is in correctly inferring the user's intent and effectively communicating those inferences back to the user.

In this thesis, I address important challenges in authoring UI automation macros by leveraging user-provided demonstrations and parameters, and structural patterns in the UI to infer generalized automation; and in understanding UI automation macros by (a) highlighting selected elements on the target UI, (b) visualizing high-level behavior through sequences of actions and UIs visited, (c) visualizing generalizations through color-coding UI elements and grouping corresponding UIs, and (d) providing feedback on validity and uniqueness of element selection logic. First, I conducted two studies observing how programmers write automation code. One of the key challenges participants experienced was in identifying appropriate UI element selection logic. Next, I designed two programming-by-demonstration systems, ParamMacros and ScrapeViz, that enable users to create automation macros without writing code. Users provide demonstrations of what UI elements they want to click or scrape, and then these systems leverage structural patterns in the website DOM to identify patterns and infer generalized automation. ParamMacros supports parameterized macros (powered by user-provided parameters) while ScrapeViz supports distributed hierarchical web scraping macros. ScrapeViz also provides visual tools to help users understand automation behavior in the context of the page source and across different UI pages. This thesis contributes learnings about the challenges users face in creating UI automation macros, and no-code authoring tools and visual understanding tools which have the promise to make UI automation more accessible to a wider audience.

CHAPTER 1

Introduction

Our devices offer vast amounts of knowledge and services through their user interfaces (UIs) – web browsers and desktop applications on computers, apps on smartphones and smartwatches, car touchscreens, and more. This provides great access for end-users, but sometimes certain information can still be difficult to access or functions tedious to perform by hand.

UI automation is a technique that enables users to programmatically perform UI tasks (rather than needing to perform them by hand) to save time or effort or to work around accessibility issues. UI automation programmatically mimics a user's interactions with UI widgets, e.g., clicking a button, typing into a text field, or copying text from a page. Users trigger a UI automation macro to run typically by clicking a button, issuing a command to a virtual or voice assistant, or running a script. UI automation can be useful for tasks that the user expects to perform many times (e.g., checking if concert ticket sales are open yet), for tasks that are especially tedious (e.g., navigating through multiple menus to change a common software setting), for tasks that are long-running (e.g., scraping article headlines from an online newspaper multiple times each day), for enabling voice or natural language-based control to make UIs more accessible for the visually impaired [51, 103], and for testing software [35, 14, 54]. UI automation is especially useful when the desired data or actions are not available in a public database or application programming interface (API) [43] and are only available on public website pages themselves. For example, an individual trying to collect article headlines from the New York Times[1] would be able to query the New York Times's API [27] to retrieve such data, but smaller, more regional newspapers such as the Baltimore Sun[2] do not offer public APIs or databases – the individual would need to laboriously copy article headlines from the newspaper website manually, or they could instead leverage UI automation. Beyond personal productivity and accessibility needs, UI automation can also help social scientists and investigative journalists conduct important large-scale data collection efforts [45], for example to track COVID cases and vaccinations [12], political ads [2] and extremist groups [39] on Facebook, government legislation [30, 10], illegal marketplaces [5, 28], and housing inequalities [24, 74].

[1]https://www.nytimes.com/
[2]https://www.baltimoresun.com/

Although UI automation proves very powerful, most often UI automation that meets a user's exact needs will not already exist – users will instead need to create their own custom automation, which can be challenging. Even if a user manages to find a relevant UI automation browser extension or script online, it may not perform the exact behaviors the user wants or may not work on the desired website. Traditionally, people have created automation macros by writing code, for example using popular libraries like Selenium [35], Puppeteer [31], Playwright [29], Cypress [14], or Beautiful Soup [8] which provide APIs for interacting with web browsers, UIs, and UI elements. However, people still must still supply the logic for leveraging these UI interaction APIs. Since people are typically not the creators of the target UI they want to automate, they need to explore how the UI works and interpret the UI's implementation. For example, the UI of a website is represented by its Document Object Model (DOM) [16] – the DOM is a tree hierarchy of nodes representing elements on the page, their relationships with each other, and spatial and stylistic properties. To perform an operation such as clicking on a button, the macro author needs to first 1) locate the desired button within the website's DOM (e.g., using the browser's "Inspect" developer tool), 2) identify appropriate XPath [46], CSS selector [13], and/or JavaScript [44] logic to select that element, and finally 3) write code to perform the desired operation (e.g., "click"). Macro authors may then want to include a variety of other logic such as repeating an operation over multiple similar elements (e.g., scraping article headlines from a newspaper website), conditionally performing an operation, determining an operation based on some given input, and so on. Handwriting UI automation code offers macro authors full control but requires a certain level of programming expertise and familiarity with UI technologies.

To make UI automation more accessible to people with limited programming experience, researchers have explored *programming-by-demonstration* (PBD) [62, 93] as an approach to creating macros [95, 87, 88, 90, 91, 89, 55, 110, 67, 64, 106, 104, 107, 57]. With PBD, the user provides a small number of *demonstrations* of how they want their program to behave in specific situations, and then the system infers how the program should behave broadly. Demonstrations can take a variety of forms, for instance, program input-output examples or a sequence of user interaction events on a UI – PBD in the context of UI automation leverages the latter. A key challenge of PBD is correctly inferring user intent. Prior PBD systems have been created for personal task automation – CoScripter [95, 87] turns a user's sequence of UI interactions into a pseudo natural language script, and Sugilite [88] and its follow-on work [90, 91, 89] help users create custom automation to extend voice interfaces on their mobile phones. Other PBD systems have been created specifically for web scraping – Rousillon [55] and WebRobot [64] and its follow-on work [106, 107, 57] all leverage patterns in the DOM to infer a generalized macro from demonstrations.

My dissertation demonstrates the following thesis statement:

We can address important challenges in authoring user interface automation macros by

leveraging user-provided demonstrations and parameters to identify meaningful structural patterns in UI implementation and infer generalized automation logic; and in understanding user interface automation macros by (a) conveying macro-selected UI elements through highlighting them on the target UI, (b) visualizing high-level behavior through sequences of actions and UIs visited, (c) visualizing generalizations through color-coding of comparable UI elements and showing corresponding UIs side by side, and (d) providing feedback on validity and uniqueness of element selection logic.

In this dissertation I explore UI automation specifically in the context of websites and Web technologies, and accordingly will use the term *web automation* as I describe my work. However, I believe many of the findings would translate to other kinds of UIs. First, in chapter 3, I design a prototype web automation IDE and study the challenges that programmers face when writing web automation scripts with this IDE and other programming environments. Many of the challenges programmers experienced were related to identifying appropriate element selection logic. Next, to address challenges with element selection, I design two programming-by-demonstration (PBD) systems, ParamMacros (chapter 4) and ScrapeViz (chapter 5), where users are able to select desired elements and create automation macros without writing code. In both systems the user provides interaction sequence demonstrations, and the inference engine leverages structural patterns in the website DOM to infer a generalized program. In ParamMacros, the user provides a single demonstration sequence and also generalization hints in the context of their program input – how their input natural language query can be parameterized. In ScrapeViz, the user provides two examples for each kind of action they want to perform – the system then generalizes to select other elements on the page that should be treated similarly. Additionally, the earlier studies I conducted explore the challenges programmers face in understanding their macro code in the context of the target UI, especially in debugging unexpected automation or scraping behavior. I briefly explored no-code program descriptions for understanding inferences in ParamMacros but found them to be too abstract for most people. This in part inspired my work with ScrapeViz which balances no-code abstractions with concrete UI examples in its program representations. ScrapeViz provides users a visual representation of the sequence of actions performed and website pages visited, and tools for understanding scraped data in the context of their page source.

I contribute the following:

1. *A web automation IDE and user studies exploring the challenges and needs of programmers writing web automation scripts (Chapter 3).* We conducted two studies with JavaScript programmers – one with participants writing web automation scripts in a traditional text editor, and one with participants writing web automation scripts in environments that provide UI context and feedback (including a web automation IDE I built). We uncovered a number of challenges, one of the most salient being identifying appropriate element selection logic.

Based on these findings, we present design implications for future web automation tools.

2. *ParamMacros, a PBD system for helping users create macros that answer parameterized questions about website content (Chapter 4).* ParamMacros leverages end-user programmers' expertise about their domain of interest. With ParamMacros, the user provides a concrete question they want to answer about content on a specific website. The user then identifies parameters in their question – i.e., parts of the question that can be replaced with different values – based on variations of the question that they expect they might want to ask in the future. The system then proposes alternative values for those parameters, which users can edit. Finally the user provides a demonstration of how to answer a specific instance of the parameterized question. ParamMacros's inference engine then infers a generalized macro by considering the user-provided parameters and leveraging structural patterns in the website DOM. In a lab study participants identified meaningful parameters in questions and successfully created generalized macros.

3. *ScrapeViz, a direct manipulation and visualization system for authoring and understanding distributed hierarchical web scraping macros (Chapter 5).* ScrapeViz is a PBD tool for authoring distributed hierarchical web scraping macros, focused on providing users a visual representation of their macro. The user provides demonstrations of navigation and scraping actions on target website pages. By giving two example elements for each action, the user conveys their desired generalization. The system then infers other elements to generalize to, and illustrates generalization through a visual representation: a storyboard that shows the sequence of pages visited, in-page highlighting to indicate elements that are clicked or scraped, color-coding to show elements that are treated similarly, and groups of similar kinds of web pages visited. ScrapeViz also provides an interactive table to link scraped data with their source website pages. In a lab study comparing to Rousillon [55], participants found ScrapeViz's visual representation helpful for understanding the macro at a high-level and the interactive table for understanding scraped data in the context of their source website pages.

A summary of these contributions is provided below in Figure 1.1.

Outline for the rest of this thesis: In chapter 2, I describe related work. In chapter 3, I describe the web automation IDE I built, the studies I conducted with programmers, and the challenges they experienced. In chapter 4, I describe the ParamMacros PBD system I designed. In chapter 5, I describe the ScrapeViz authoring and visualization tool I built. Then in chapter 6, I discuss learnings, limitations, and implications of my work, and finally in chapter 7, I provide concluding thoughts.

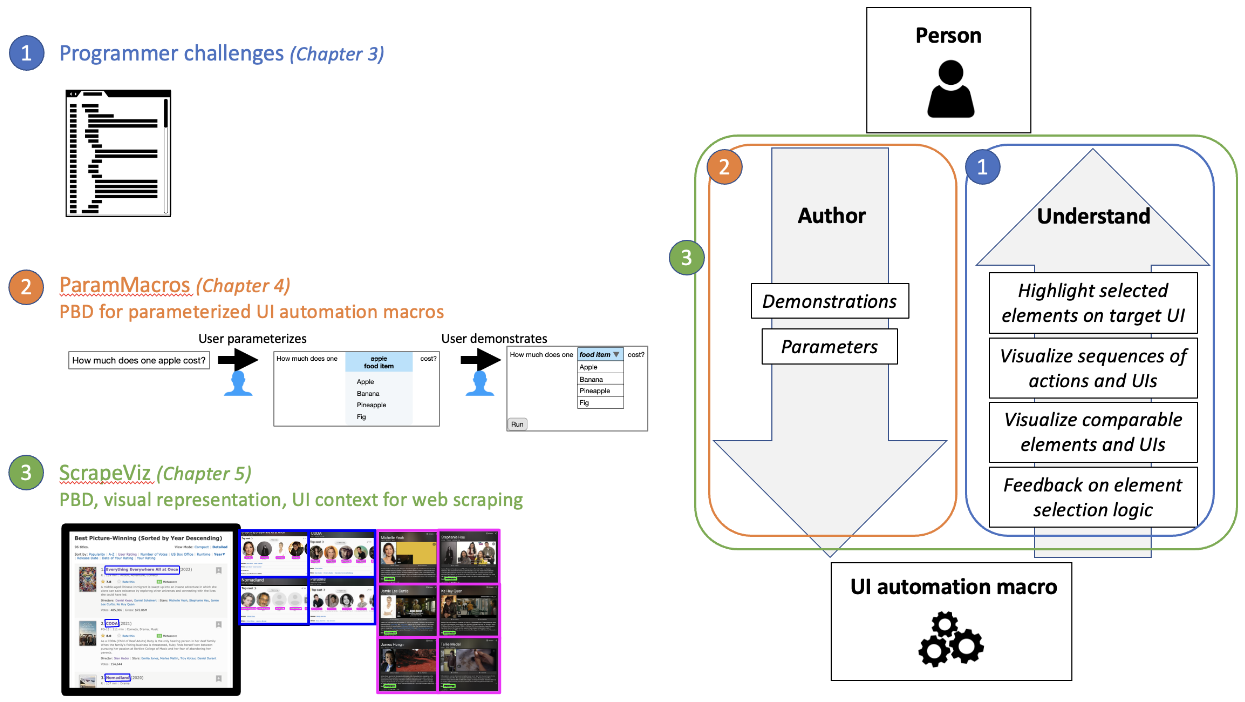

1 Programmer challenges (Chapter 3)
2 ParamMacros (Chapter 4)
PBD for parameterized UI automation macros
User parameterizes
How much does one apple cost?
User demonstrates
food item
Apple
Banana
Pineapple
Fig
Run
3 ScrapeViz (Chapter 5)
PBD, visual representation, UI context for web scraping
Person
Author
Understand
Demonstrations
Parameters
Highlight selected elements on target UI
Visualize sequences of actions and UIs
Visualize comparable elements and UIs
Feedback on element selection logic
UI automation macro

Figure 1.1: This thesis addresses challenges in authoring and understanding UI automation macros. (1) Chapter 3 identifies challenges and needs of programmers writing UI automation macros. (2) Chapter 4 presents ParamMacros, a programming-by-demonstration (PBD) system for creating parameterized UI automation macros without code. (3) Chapter 5 presents ScrapeViz, a PBD system for creating distributed hierarchical web scraping macros and which provides users visual tools for understanding automation behavior.

CHAPTER 2

Background[1]

This thesis draws on related work in UI automation programming tools, programming by demonstration, direct manipulation interfaces, UI context in developer tools, program visualization, and natural language interfaces.

2.1 Background on UI Automation

User interface (UI) automation is useful for saving time and energy on tedious and repetitive computer tasks (e.g., approving employee payroll, scraping data), testing software systems robustly at scale, and automating web tasks on inaccessible websites for blind users [103, 51, 52, 50]. Scaffidi et al. [109] outline observed scenarios of users manually interacting with websites that would likely benefit from web automation, and requirements for web automation tools to support these. We observed many of these needs in our study with programmers writing automation macros (chapter 3) and while we were designing ParamMacros (chapter 4) and ScrapeViz (chapter 5) ourselves.

2.2 Writing UI Automation Scripts By Hand

Developers can write custom automation scripts that programmatically mimic a user's interactions on a user interface (UI). In most web automation frameworks, programmers write code that simulates interactions such as clicking and typing in a web browser. In order to specify which UI elements to interact with, programmers typically use XPath [46] or CSS selectors [13]. Both XPath and CSS selectors reference the DOM (Document Object Model) [16]—a tree structure that represents page content. The typical setup for writing web automation scripts consists of a code editor

[1]Parts of this chapter are adapted from the Related Work sections of these publications: **1)** Rebecca Krosnick and Steve Oney. Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2021).* **2)** Rebecca Krosnick and Steve Oney. ParamMacros: Creating UI Automation Leveraging End-User Natural Language Parameterization. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2022).*

(for writing the automation code) and a web browser with developer tools [19, 17] (for referencing the page's DOM).

Popular frameworks include Selenium [35], Puppeteer [31], Playwright [29], Cypress [14], and Beautiful Soup [8] for the web, and Shortcuts [37] and App Actions [6] for mobile. All these frameworks work similarly—programmers write code in these frameworks (which provide functions for simulating user input, referencing the page, and more). Most of these frameworks simply show the real-time execution of the script on the website UI. Cypress is a newer framework that additionally allows the programmer to post hoc inspect the page state before or after any script command and see which elements were selected. We conducted two lab studies to understand the challenges programmers face writing web automation scripts (chapter 3) – in the first study we observed participants using traditional text-based editors and in the second study we observed participants using environments that provide UI context and feedback (Cypress and a prototype IDE we built). One of the more salient findings was that it can be challenging to construct UI selectors [13] that are robust across different contexts and inputs.

Researchers have proposed making UI automation easier by simplifying the language used to write web automation macros. Koala [95] and CoScripter [87] represent scripts in a language that is close to natural language—for example, "Click 'Add to cart'." Similarly, Sikuli [125] allows programmers to specify elements visually (with screenshots) for desktop-based automation. With these tools, developers would not need to reference the page's internal DOM structure. Instead, the interpreter searches the page for an element that fits the high-level description of the target element. However, scripts generated in these systems are often not as expressive (because the language is limited) or robust as scripts that explicitly reference the internal page DOM.

2.3 Record and replay tools

Record and replay tools like Selenium IDE [36], iMacros [22], and Cypress Studio [15] were designed for test automation and can generate code from a single user trace, but the code will not be generalized to work across scenarios. ParamMacros (chapter 4) and ScrapeViz (chapter 5) enable users to create generalized macros without writing code.

2.4 Programming by Demonstration

2.4.1 Background

Programming by demonstration (PBD) [62, 93] enables end-users to create computer programs without writing code – instead of writing code, users just provide concrete demonstrations or examples of the desired behavior. Demonstrations may take one of a number of forms, for instance a sequence of user actions, a sequence of program or application states, or simply program input-output examples (more commonly called programming by example (PBE)). A key challenge of PBD is inferring user intent and generalizing from demonstrations. PBD has a rich history, with systems that support UI creation [97, 68, 81], text and code editing [85, 99, 101], data transformation [71, 65], constructing regular expressions [53, 126], creating visualizations [116, 129], and more.

2.4.2 PBD for web scraping

Prior work has explored using PBD for creating web scraping scripts. Rousillon [55] and WebRobot [64] can synthesize nested loop-based scraping logic from user demonstrations by leveraging patterns in the DOM [16]. ParamMacros and ScrapeViz similarly leverage patterns in the DOM.

Rousillon, WebRobot, and our ScrapeViz system all synthesize nested-loop web scraping macros but take slightly different forms of user input. Rousillon accepts a single demonstration of the first "row" of data it should scrape, and then infers a generalized macro for scraping the rest of the rows. Meanwhile, WebRobot accepts an iterative sequence of demonstrations, essentially demonstrations of scraping multiple rows of data, and looks for a generalized scraping pattern to infer a macro. WebRobot then proposes the next action, which the user can either accept or decline. ScrapeViz takes an approach more similar to WebRobot, generalizing once the user has provided two example UI elements that have a common XPath formula. SemanticOn [106] builds on WebRobot to also support users in specifying conditional scraping logic through natural language (supported through language models).

Rousillon, WebRobot, and ScrapeViz all provide user interfaces for understanding how their macro works. Rousillon and ScrapeViz both use color-coding to indicate corresponding UI elements that will be treated the same way (e.g., all movie titles in blue, all movie years in pink), but Rousillon does this only during authoring (before users scrape) while ScrapeViz provides this color-coding for users to understand scraping generalizations after the fact. WebRobot does not specifically convey generalization visually. WebRobot, however, during authoring mode does provide the user a sneak peak of how their macro will behave next, i.e., the next item it predicts scraping, which the user can accept or correct. Rousillon also presents the user a block-based program. ScrapeViz combines color-coding with a storyboard visual of the sequence of pages visited and corresponding pages as groups to provide a broad overview of macro behavior across page contexts. ScrapeViz also provides an interactive table to enable users to understand scrape data in the context of their page source. MIWA [57] builds on WebRobot and offers users a natural language description of the sequence of scraping actions in their generated macro. This description is also interactive, so users can hover over particular items and see the corresponding UI elements from the website page highlighted. ScrapeViz does not provide a natural language description of the scraped data, but does allow users to see data scraped into the output table in the context of their page source.

2.4.3 PBD for personal task automation

Personal task automation is another domain with a rich PBD history. CoScripter [87, 95] lets users record their actions on the web and generates a pseudo-natural language script. CoScripter users can create a personal data store containing personal information (e.g., name, email) so that the generated script uses parameters in their place, which is important when shared with colleagues. CoScripter focuses on form-filling and uses parameter values to fill in form fields. ParamMacros uses parameters to generalize dynamic element selection.

Most similar to ParamMacros is the Sugilite suite [88, 90, 91], which enables end-users to create custom automation for responding to speech requests and completing tasks on their mobile device. With Sugilite [88], users provide a Natural Language (NL) request and a demonstration of the actions to complete that request. Sugilite then infers parameters and a generalized program to work over the different parameter values. Sugilite infers parameters by searching for features of a given UI event (e.g., text typed into a text field, label of a clicked button) within the NL request. If a parameter is identified, our understanding is that Sugilite then searches for alternative parameter values by looking at the target UI element's sibling nodes. Appinite [90] extends Sugilite using NL understanding (NLU) and an improved understanding of the UI. Ahead of inference, it traverses its app view hierarchy and builds a UI semantic and spatial relational knowledge graph, which it uses to better understand what elements in the UI the user's NL request is referring to at inference time. Pumice [91] extends Sugilite to support conditional logic. A key difference between ParamMacros and the Sugilite suite is that our system leverages user-provided parameters and values to identify relevant patterns in the DOM, whereas Sugilite primarily considers sibling nodes, and Appinite uses NLU to identify relevant relationships in its UI knowledge graph. Complex relationships exist between elements at many levels in a UI hierarchy, and ParamMacros and the Sugilite suite take different approaches to identifying those relationships.

DiLogics [107] is another PBD that supports web automation for diverse natural language requests for a website. The user provides a list of natural language requests and DiLogics segments them into meaningful structured tasks. For a given request, DiLogics presents the user with a sequence of subtasks to complete, and the user demonstrates these on the website. The user continues demonstrating for the next requests and DiLogics tries to identify patterns to align the subtasks with demonstration sequences, and as appropriate proposes automation to the user. ParamMacros similarly tries to identify parameters in a user's demonstration to generalize it to other inputs, but relies on a single user demonstration rather than multiple.

AutoVCI [104] and VASTA [110] are two other single-demonstration PBD systems for creating automation for speech requests. Similar to Sugilite, both automatically identify potential parameters by mapping text in a user's natural language request to UI elements from the demonstration interaction sequence. Unlike Sugilite and ParamMacros, AutoVCI asks the user a sequence of strategic yes/no questions to help clarify the appropriate app, actions, and parameters. VASTA uses computer vision to identify from a UI screenshot the appropriate UI elements to interact with, instead of programmatically interacting with the UI's view implementation.

DIYA [67] also leverages PBD for generating parameterized web automation for natural language requests. With DIYA, the user speaks as they are interacting with a website UI to describe and demonstrate values to extract. For example, would could a macro that computes the price on a particular website page by first speaking "Start recording price", then selecting the price value on the website page with their mouse and speaking "Return this value", and then speaking "Stop recording". Similar to ParamMacros, DIYA also relies on the user to identify parameters, though the interaction is different. For instance, for a user trying to create a macro for computing the cost of chocolate chip cookie recipe, the user would define a parameter "recipe" after they have typed "chocolate chip cookies" into a searchbar by speaking "This is a recipe". The user identifies parameters by speaking them as they demonstrate, different from ParamMacros where the user up front identifies parameters in their text-based request and then later demonstrates. Note that DIYA's contribution focuses on interactions and an associated programming language to support function composability, iteration, and conditionals, and as a result uses simpler element selection inference logic than ParamMacros, relying on website pages to use semantically meaningful IDs [21] and classes [9].

Etna [108] collects user interaction traces on a website over time, essentially enabling it to work with multiple demonstrations to identify common automation logic and parameters. ParamMacros instead uses only a single demonstration and relies on the user to explicitly specify parameters instead of trying to guess them.

Savant [49] generates task shortcuts for user NL requests – it maps a user's NL request to the best-matching app screen from the Rico dataset [63] and fills in textfields based on parameters in the NL. With Savant, the possible task shortcuts that can be created are based on the apps and interaction traces available in the Rico dataset, and the parameters the researchers manually defined. In contrast, ParamMacros can potentially support previously unseen UIs and automation tasks because it relies on the end-user to provide a demonstration and parameters.

2.5 Direct Manipulation as Specification

Rather than using an example- or demonstration-based approach, other prior work has leveraged direct manipulation to directly specify behavior. This typically falls into one of two camps, or a combination of the two: directly modifying the target, or creating annotations on the target.

In bi-directional code and UI output environments, the user can edit either the code or the UI to update the program – the code and UI remain synced. In "Inventing on Principle" [115] and Sketch-in-Sketch [72, 73] the user can add visual elements, edit styles and properties, and set constraints by directly manipulating the target illustration or SVG. In mage [80], the user can directly manipulate widgets, for example reordering columns in a table, and corresponding code is automatically generated.

In Kitty [77], Draco [78], and other environments for authoring dynamic and interactive illustrations [79, 123, 117, 122], the user draws annotations and selects properties on overlaid menus to specify desired animations and data bindings. Others have explored direct manipulation interfaces that enable artists to create constraint and state-based logic in their procedural generative art [76, 75].

Our ScrapeViz system also leverages direct manipulation on its target UI, but through demonstrations of data to scrape rather than a logical specification.

2.6 Context, Feedback, and Representations

2.6.1 UI Context and Feedback

As we describe in chapter 3, many of the challenges of writing web automation code can be categorized as the need for UI context or live feedback. Although the specific context and feedback needs for web automation developers are unique, prior research has explored mechanisms for integrating context and feedback into development tools.

Some programming systems [100, 98, 59, 60] generate storyboards [114] to illustrate the sequence of program actions and their resulting user interface states. Cypress and our web automation IDE prototype that we evaluated in Study 2 (chapter 3) similarly offer UI snapshots to explain program behavior. Though not for UI programming, projection boxes [86] compactly illustrate variable values per line of code and across loop iterations, which our web automation IDE prototype similarly does for UI states.

Kubelka et al. [83] studied the kinds of immediate feedback features programmers use in several languages, including JavaScript. They observed how programmers heavily use the DOM inspector and console to get faster feedback. In our work, we similarly observed how programmers heavily use the DOM inspector and console. We also observed challenges programmers face specific to web automation and then evaluated environments that offer continual or live feedback on UI state sequences and UI element selection.

2.6.2 Visual Representations

Prior work has leveraged storyboard and data flow visualizations for helping users understand automation macros and UI behavior. DemoScript [60] automatically generates storyboards [114] to illustrate device states and operations in cross-device interaction scripts. ScrapeViz similarly generates a storyboard visualization, but also expresses generalized behavior through UI element color-coding and groups of pages. Marmite [118] presents a data flow programming environment where authors combine operators in sequence to program extraction and processing of data from web pages. ScrapeViz and Marmite both present a storyboard-like visualization of their programs, as well as an output spreadsheet. Showing results in the context of target website pages is beneficial because it can help users understand whether their program is extracting the correct data from the page. ScrapeViz also shows extracted data in the context of the spreadsheet through color-coded borders.

2.7 Natural Language Interfaces

2.7.1 Virtual Assistants

Virtual assistants like Siri [38], Alexa [4], Google Assistant [20], and Cortana [11] have become commonplace, and are powerful because they enable hands-free interaction. Each virtual assistant has a built-in set of common skills it supports, but there are endless complex or obscure requests this does not include. Our system ParamMacros enables end-users to build question-answering programs, that potentially could be useful to virtual assistants, without needing to write program code.

2.7.2 Data and Models for UI Automation

Prior work has also explored natural language processing approaches for interpreting a user's natural language requests and performing automation on a user interface. In [92], the authors collect datasets of user natural language requests and the corresponding actions that should be performed on a mobile UI. They then train transformers to extract relevant language and UI properties and then ground the language in the UI. FLIN [96] explores a semantic parser approach to map a user's natural language to the most relevant high-level conceptual action on the given website. Adept has also recently built a transformer, ACT-1 [1], for converting natural language into UI automation. ParamMacros leverages built-in heuristics and user-provided custom demonstrations rather than models trained on large datasets.

2.7.3 Question-Answering Systems

Question-answering systems [128] take a user's natural language query as input, identify potentially relevant documents (e.g., websites on the web), and then search through those unstructured documents to find the best answer. Although these AI techniques are powerful, there will be situations where they do not produce the answer the user wants. ParamMacros allows users to create custom automation for their specific needs that are not met by an existing machine learning model.

2.7.4 Natural Language and Data

Natural language interfaces to databases (NLIDBs) [47] enable end-users to query databases without needing to understand structured query languages like SQL. NLIDBs inherently only support answering questions about data that is already structured. ParamMacros helps end-users create custom automation on-demand when there is no database already.

Prior work also explores natural language interfaces for data visualizations [111, 112, 70]. DataTone [70] is an NLIDB that allows flexibility for ambiguous natural language queries. The system identifies tokens in the NL that it thinks are ambiguous and their possible interpretations in the context of the database. DataTone offers a parameter-based UI (a parameter per token) and allows the user to select parameter values to run their query on, similar to ParamMacros's UI.

CrossData [58] identifies relationships between a writer's prose and embedded tables and charts – automatically extracting data values and allowing writers to explore alternative properties. CrossData identifies parameters and values in prose automatically using NLP techniques, whereas in ParamMacros we ask users to identify parameters themselves.

CHAPTER 3

Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts[1]

3.1 Introduction

The Web is a rich source of information and services. The vast majority of web content was designed to be accessed by people through web browsers. However, there is tremendous value in providing services that are also computer-accessible. *Web automation macros*—programs that mimic human input to interact with web pages—can help users perform repetitive tasks, test software applications at scale, help users overcome web accessibility issues, and more. Decades of research into web automation has explored how to allow computers to extract information [61] and perform actions [88, 95, 87] on web content. Although many tools for web automation have been proposed, the fundamental challenges of writing web automation code remains understudied.

The particular challenges and needs of web automation tools are important to understand for several reasons. First, web automation (and related techniques like Robotic Process Automation) are increasingly common as more information and services continue to be digitized. Second, an evidence-backed description of the challenges of web automation can help provide valuable design guidelines to a large and growing body of work into web automation tools. Finally, several aspects of writing web automation code make it meaningfully different from other kinds of programming. Writing web automation code requires referencing an external data source (a web page) that was designed to be consumed by humans, rather than code. Aspects of interacting with a web page that are second-nature to people—referencing a particular button, handling unexpected content, and dealing with sequentiality and timing—can be challenging to deal with in code. Further, web pages change over time (e.g., through redesigns or internal refactoring) and change with context (e.g., with A/B testing).

[1]This chapter is adapted from the publication: Rebecca Krosnick and Steve Oney. Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2021).*

This paper contributes an evidence-backed description of the challenges of writing web automation macros. We conducted two studies—one focused on the general challenges of writing web macros and another focused specifically on providing feedback and context—to better understand these challenges. Among other things, our findings include that developers need feedback and UI context about the page elements they are selecting and interacting with. This paper contributes:

- A first study, uncovering the general challenges programmers face when writing web automation scripts in a traditional text editor.
- A web automation IDE prototype that presents UI snapshots and feedback on element selection across multiple execution contexts.
- A second study, understanding where UI feedback and context features can help programmers writing web automation scripts, and where support is still lacking.
- Design implications for future web automation tools.

3.2 Study 1: Traditional Editor Environment

We conducted a user study to learn the strategies programmers use and the challenges they encounter when writing web automation scripts in a traditional text editor environment.

3.2.1 Study Design

We recruited 15 participants (3 female, 12 male; 20–40 years old) from our university and social media. All participants reported substantial experience with JavaScript and querying the DOM with CSS selectors. Six had 2–5 years and nine had at least 5 years of general programming experience. Our participants included eight professional developers, one product designer, five graduate students, and one undergraduate student. All but one participant reported at least some prior experience with creating web automation scripts.

Each session lasted 90 minutes and participants were compensated with a $25 USD (or equivalent) Amazon gift card. We asked participants to use Puppeteer [31] to write a web automation script. Only one participant had prior experience using Puppeteer. We first gave participants a 15 minute tutorial to familiarize them with Puppeteer. During the task we gave participants reference material for Puppeteer and CSS, allowed them to search online, answered questions about syntax, and provided hints if they were stuck for awhile. We gave each participant one of three tasks to work on for 45 minutes (each task was assigned to five participants):

- *Airbnb* or *Google Hotels*: Create a script that searches for hotels. Set a location (text field), check-in and check-out dates (calendar widget), and display matching results.
- *Amazon*: Create a script for identifying an item to purchase. Search for an item (text field), indicate it must be available via Prime (checkbox), find the first result with a "Best Seller" label, and print out the name of the item.

We chose these tasks to observe a variety of scripting steps participants would need to take (e.g., advancing a calendar to the desired month on Airbnb and Google Hotels; querying for appropriate ancestor and descendant DOM elements on Amazon), as well as their element selection strategies for a variety of website DOMs. Although the Airbnb and Google Hotels tasks are semantically very similar, we used both because we noticed the Google Hotels DOM is complex and many participants were stuck in the early stages of the task.

We asked participants to generalize their scripts to support variable input values (i.e., locations, dates, item to purchase) and gave them two test cases to ensure their script worked for. We then conducted a brief interview.

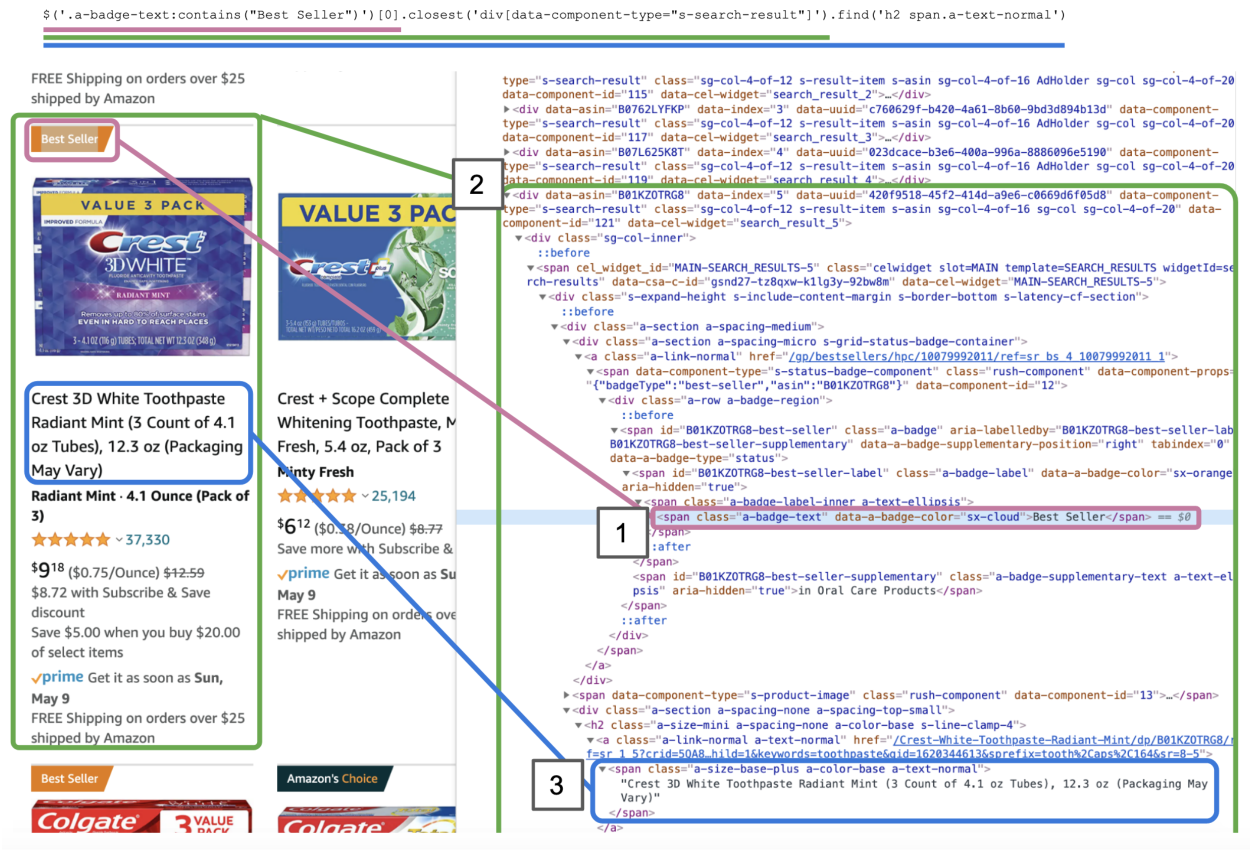
3.2.2 Findings

3.2.2.1 Selecting UI elements

In order to correctly select a desired UI element, participants had to choose CSS selectors [13] that **uniquely** identify the desired element and are **robust** to page state changes and varying user input. This involved inspecting the DOM to understand the relationships between nodes and reasoning about selector specificity, either based on intuition or by testing selectors manually. For many element selection subtasks, choosing appropriate CSS selectors took a few minutes and some iteration (e.g., stacking selectors once the participant realized a particular CSS class was not unique enough), but were not overly difficult. Other element selection subtasks were more challenging, as we describe below:

Sometimes a unique identifier is not robust across sessions. Some websites (e.g., the Google Hotels website in our study) randomly generate the letters/numbers in IDs [21], classes [9], or attributes [7] per page load or browser session. However, some participants did not realize this ahead of time and accidentally chose selectors containing randomly generated strings. Two Google Hotels participants did this when trying to select the calendar element, using the dev tool's "Copy selector" feature to get a unique selector for it (e.g., `#ow28 > div:nth-child(1) > div:nth-child(1) > div:nth-child(1) > div:nth-child(2)`), but this included an ID (e.g., `#ow28`) that was randomly generated per page load. These participants were then puzzled when the selector did not match any elements on the next run, and when they tried

"Copy selector" again and got a different selector this time (e.g., `#ow24 > div > div > div:nth-child(1)`). C3 spent 20 minutes and C5 ten minutes unsuccessfully investigating why their selectors were not working before the study facilitator explained that the IDs change per page load.



```
$('.a-badge-text:contains("Best Seller")')[0].closest('div[data-component-type="s-search-result"]').find('h2 span.a-text-normal')
```

FREE Shipping on orders over $25
shipped by Amazon

Best Seller

Crest 3D White Toothpaste Radiant Mint (3 Count of 4.1 oz Tubes), 12.3 oz (Packaging May Vary)
Radiant Mint · 4.1 Ounce (Pack of 3)
37,330
$9.18 ($0.75/Ounce) ~~$12.59~~
$8.72 with Subscribe & Save discount
Save $5.00 when you buy $20.00 of select items
prime Get it as soon as Sun, May 9
FREE Shipping on orders over $25 shipped by Amazon

Crest + Scope Complete Whitening Toothpaste, M
Fresh, 5.4 oz, Pack of 3
Minty Fresh
25,194
$6.12 ($0.38/Ounce) ~~$8.77~~
Save more with Subscribe &
prime Get it as soon as Su
May 9
FREE Shipping on orders ove
shipped by Amazon

Best Seller
Amazon's Choice

```
type="s-search-result" class="sg-col-4-of-12 s-result-item s-asin sg-col-4-of-16 AdHolder sg-col sg-col-4-of-20
data-component-id="115" data-cel-widget="search_result_2">…</div>
▶<div data-asin="B0762LYFKP" data-index="3" data-uuid="c760629f-b420-4a61-8b60-9bd3d894b13d" data-component-
type="s-search-result" class="sg-col-4-of-12 s-result-item s-asin sg-col-4-of-16 AdHolder sg-col sg-col-4-of-20
data-component-id="117" data-cel-widget="search_result_3">…</div>
▶<div data-asin="B07L625K8T" data-index="4" data-uuid="023dcace-b3e6-400a-996a-8886096e5190" data-component-
type="s-search-result" class="sg-col-4-of-12 s-result-item s-asin sg-col-4-of-16 AdHolder sg-col sg-col-4-of-20
data-component-id="119" data-cel-widget="search_result_4">…</div>
▼<div data-asin="B01KZOTRG8" data-index="5" data-uuid="420f9518-45f2-414d-a9e6-c0669d6f05d8" data-component-
type="s-search-result" class="sg-col-4-of-12 s-result-item s-asin sg-col-4-of-16 sg-col sg-col-4-of-20" data-
component-id="121" data-cel-widget="search_result_5">
  ▼<div class="sg-col-inner">
     ::before
    ▼<span cel_widget_id="MAIN-SEARCH_RESULTS-5" class="celwidget slot=MAIN template=SEARCH_RESULTS widgetId=se
    rch-results" data-csa-c-id="gsnd27-tz8qxw-k1lg3y-92bw8m" data-cel-widget="MAIN-SEARCH_RESULTS-5">
      ▼<div class="s-expand-height s-include-content-margin s-border-bottom s-latency-cf-section">
         ::before
        ▼<div class="a-section a-spacing-medium">
          ▼<div class="a-section a-spacing-micro s-grid-status-badge-container">
            ▼<a class="a-link-normal" href="/gp/bestsellers/hpc/10079992011/ref=sr_bs_4_10079992011_1">
              ▼<span data-component-type="s-status-badge-component" class="rush-component" data-component-props=
               "{"badgeType":"best-seller","asin":"B01KZOTRG8"}" data-component-id="12">
                ▼<div class="a-row a-badge-region">
                   ::before
                  ▼<span id="B01KZOTRG8-best-seller" class="a-badge" aria-labelledby="B01KZOTRG8-best-seller-lab
                  B01KZOTRG8-best-seller-supplementary" data-a-badge-supplementary-position="right" tabindex="0"
                  data-a-badge-type="status">
                    ▼<span id="B01KZOTRG8-best-seller-label" class="a-badge-label" data-a-badge-color="sx-orange
                    aria-hidden="true">
                      ▼<span class="a-badge-label-inner a-text-ellipsis">
                         <span class="a-badge-text" data-a-badge-color="sx-cloud">Best Seller</span> == $0
                       </span>
                       ::after
                     </span>
                     <span id="B01KZOTRG8-best-seller-supplementary" class="a-badge-supplementary-text a-text-el
                     psis" aria-hidden="true">in Oral Care Products</span>
                   </span>
                   ::after
                 </div>
               </span>
             </a>
           </div>
          ▶<span data-component-type="s-product-image" class="rush-component" data-component-id="13">…</span>
          ▼<div class="a-section a-spacing-none a-spacing-top-small">
            ▼<h2 class="a-size-mini a-spacing-none a-color-base s-line-clamp-4">
              ▼<a class="a-link-normal a-text-normal" href="/Crest-White-Toothpaste-Radiant-Mint/dp/B01KZOTRG8/r
              f=sr_1_5?crid=5QA8…hild=1&keywords=toothpaste&qid=1620344613&sprefix=tooth%2Caps%2C164&sr=8-5">
                ▼<span class="a-size-base-plus a-color-base a-text-normal">
                   "Crest 3D White Toothpaste Radiant Mint (3 Count of 4.1 oz Tubes), 12.3 oz (Packaging May
                   Vary)"
                 </span>
               </a>
```

Figure 3.1: To query the Amazon DOM for the first "Best Seller" and get the item's name, (1) query for the first "Best Seller" label, (2) query for its ancestor that represents the full item, and then (3) query for the item name.

Multi-part queries through the DOM hierarchy. To select the item name for the first "Best Seller" on an Amazon results page, it was not possible for participants to query for simply a single selector. The "Best Seller" label (Figure 3.1, box 1) and the item name (Figure 3.1, box 3) are neither ancestors nor descendants of one another, but rather both descendants of a common DOM node ancestor (Figure 3.1, box 2) representing the item as a whole (which contains the "Best Seller" label, the item name, item image, etc). Four participants took the approach of first searching for the first "Best Seller" label on the page, then querying up through the DOM tree for the node representing the item as a whole, then querying down through this node's descendants to find the item name (Figure 3.1). For example, one participant's query was `$(".a-badge-text:contains(Best`

`Seller)") [0].closest('div[data-component-type= "s-search-result"]').find("h2 span.a-text- normal")`. This was non-trivial, because it required careful search of the DOM to find a common ancestor for the "Best Seller" label and item name nodes. These four participants took between 7.5 and 22 minutes to investigate, identify, and test their full selector query chain, a testament to the challenge. The fifth Amazon participant took a slightly different approach, first selecting all of the item containers on the page, then looping through to find the first one containing the text "Best Seller", and then planned to query down through this node's descendants to find the item name. This participant spent 13 minutes on this, but ran out of time.

3.2.2.2 Keeping track of DOM nodes

Most commercial websites have extensive and complex DOM trees. In order to write selectors that are correct, robust, and unique, programmers need to account for not only the target DOM node but also other elements in the DOM. For example, they might need to find the common ancestor of two elements or compare different elements to see if they have the same class. Although most browser dev tools make it easy to navigate to one particular element, they often do not help developers understand the relationships between different parts of the DOM.

3.2.2.3 Navigation and timing

Some interactions cause the browser to navigate to a different page (i.e., from the Amazon home page to a search results page). Before trying to interact with a UI element on a new page, the script needs to allow the page to finish loading (e.g., via the `waitForNavigation` [32] command). However, some participants forgot to include a "wait" command and as a result their script failed to find target UI elements on the page. It took the programmer some effort to understand why the UI element could not be interacted with, because when they manually inspect the page, they see the UI element is present.

3.2.2.4 Trouble typing into input fields

Three Google Hotels participants (C2, C3, C6) had trouble with what originally appeared to be a simple subtask – typing a location into a search bar. These participants decided to select the search bar by the selector `.whsOnd.zHQkBf` and programmatically type into it, but when they ran the script they did not see this typing behavior occur and were puzzled. One participant instead searched for a different selector to use, while the other two participants spent significant time (C2: 16 min, C6: 5 min) trying to debug, trying different things like setting the `value` attribute of the element, which also did not work as desired. The reason participants could not type into the element is because there are actually multiple `<input>` elements on the page with the same class, the first two of which correspond to the location search bar. However, it turns out that the first element is disabled, which none of the participants noticed. In order to successfully type into the search bar, participants either had to click into the first element before typing into it to give it focus, or they had to select the second element matching their selector, which turns out to not be disabled. This second element matching the selector (but not the first) has the attribute selector `[aria-label="Enter your destination"]`, which a fourth participant C4 chose on a whim at the beginning of the task and as a result never ran into the challenges the other participants faced. Relatedly, participant D3 working on the Airbnb task tried typing dates into the “Check in” and “Check out” date elements, but these elements cannot be typed into at all—the user or script has to actually click dates on the calendar. For these challenges in trying to type into or set the `value` of UI elements, participants did not receive explicit feedback from the environment that these actions could not be performed.

3.2.2.5 Interacting with calendar widgets

A large part of the Google Hotels and Airbnb tasks involved appropriately interacting with and querying the calendar widget. The calendar widget only shows two months at a time (the current month and the next month), so if trying to book a hotel for several months in the future, the script will need to advance the calendar to the correct month. Participants had to reason about how to identify if their desired month and date were visible in the calendar, which involved understanding what the DOM looks like. For example, the Airbnb calendar widget only shows two months at a time, but the DOM actually contains four months in total at a time (i.e., the prior and next months are in the DOM but visibly hidden), which impacts the logic the user might use to correctly identify the current months. Participants also had to make sure that relevant UI rendering finished before they performed queries (e.g., that the calendar finished rendering before they queried any of its contents), otherwise the desired DOM nodes might not be present.

3.2.2.6 Feedback loop and debugging

Participants used a combination of different approaches to understand the results of their code. Six participants simply ran their full in-progress Puppeteer script each time they wanted to evaluate its behavior. The other nine participants used a combination of running the full script and executing commands in the browser dev console, shortening their feedback loop. The browser dev console gave them immediate feedback on whether their CSS selectors uniquely matched the elements they intended, whether interacting with an element had the desired effect (e.g., whether clicking on a button causes the page to navigate), and whether an element had particular attributes. In fact, one participant (Amazon task) essentially drafted his entire script in the browser dev console before adapting it to the Puppeteer environment, incrementally writing commands, observing whether they worked, and adjusting as necessary. 11 participants also inserted `console.log` print statements into their scripts to check intermediate values. Six participants used the browser debugger in order to step through their code to identify the source of a problem and be able to inspect the DOM at a particular page state.

Several participants explicitly commented that the feedback loop for evaluating whether their code worked was slow, not receiving feedback on their code until the next time they actually ran the script. For example, Google Hotels participant C4 said *"I've done a fair amount of testing and I work as a front end Dev for my job. So I'm using selectors all day long. You can see clearly how many mistakes I was making and there's nothing, there's no feedback to go 'you're being a bit of an idiot here'. The computer is terrible at that....The tests are reasonably kind of slow to run. So if you get something wrong, you have to go work out kind of what's gone wrong, that's not obvious. And then you kind of go run the tests again. And by the time you've done all that, it's like well two minutes in my life, I'm never getting back"*.

3.2.2.7 Future website changes might break scripts

Six participants noted that even if they find CSS selectors that work today, their script could break at any time if the website owner changes the website's content, layout, or DOM implementation – *"I would say probably in all cases, you just can't be sure if it's going to work tomorrow...I don't know that [selecting by text] is necessarily going to be more stable than just a test ID or class name. Because who knows what they will change first"* (D4 – Airbnb).

As a proxy for testing the robustness of participants' CSS selectors and understanding how website DOMs change over time, we searched for participants' selectors in older versions of the task websites (via the Internet Archive WayBack Machine [23]) to see if they existed there. Some selectors work for website versions from the last several years, for example the `#twotabsearchtextbox` selector for the search bar on the Amazon home page works on websites back until July 2010. However, participants' selectors for other elements do not work for earlier website versions within a year of when we ran our study (October 2020). Of the four Amazon participants who finished creating a selector to select the text for the first "Best Seller" item on the page, three participants' selectors do not work on the January 2020 version because they selected by attribute values or class combinations that previously did not exist. Of the three Google Hotels participants who finished creating a selector for clicking to open the calendar widget, two participants' selectors (`.p0RA.ogfYpf.Py5Hke` and `.DpvwYc.of9kZ`) do not work on the October 2019 version, while another seemingly obscure selector (`.eoY5cb.MphfQd.yJ5hSd`) does work. In fact, `.DpvwYc.of9kZ` actually no longer works on the current Google Hotels website as of the submission of this paper (May 5, 2021). This suggests that writing selectors that are robust across page changes is a significant challenge.

3.3 Study 2: Environments that Provide UI Context and Live Feedback

We conducted a study to evaluate the benefits and limitations of web automation environments that provide the programmer UI context and feedback. We evaluated a prototype we built (Figures 3.2 and 3.3) and Cypress [14], an increasingly popular test automation framework. First we describe each environment. Then we describe the study design and results.

3.3.1 Prototype

We designed and built a prototype IDE[2] (Figures 3.2 and 3.3) for programmers writing web automation scripts, inspired in part by Study 1. The prototype provides live feedback on CSS selectors, integrates UI context within the code editor, and helps users understand script results across different user inputs and for different pages. We built this prototype to provoke new ideas about providing UI context and feedback in web automation tools, and see to what degree programmers find them useful. The prototype includes a code editor on the left, main website view in the center, and UI snapshots which pop out from the right. Chromium dev tools are available for the main website and UI snapshots.

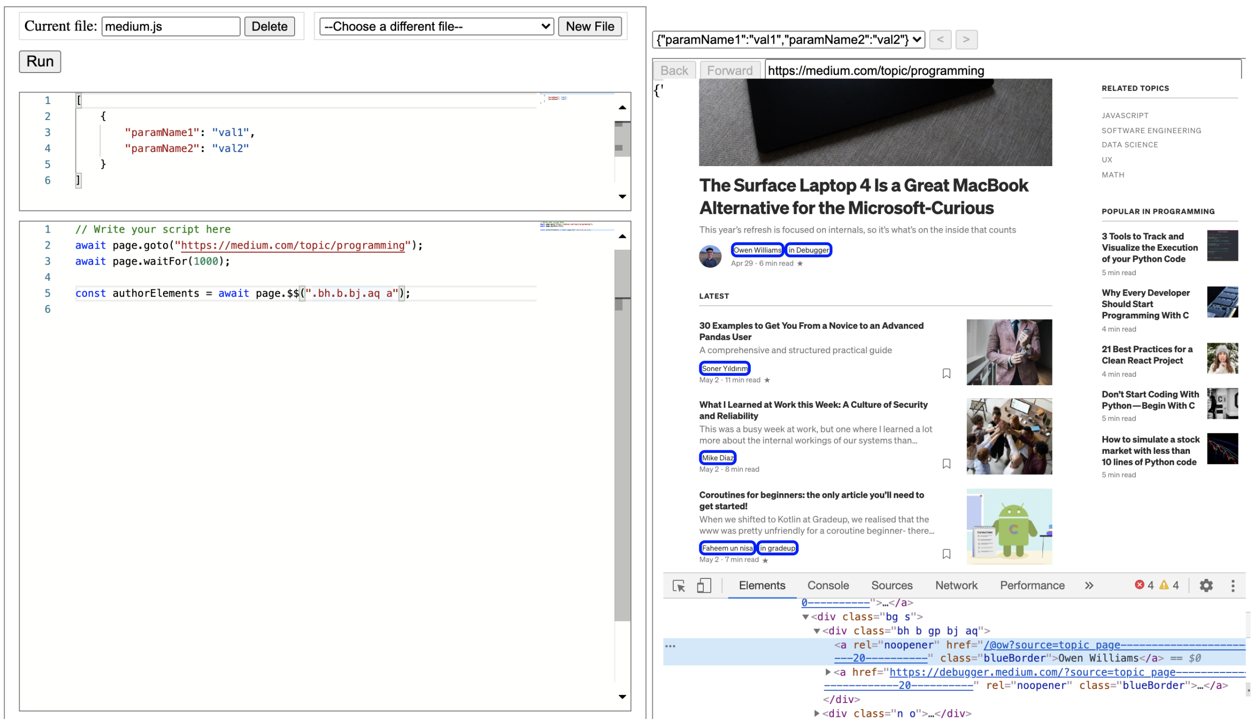
3.3.1.1 Dynamic element highlighting

When the programmer writes a CSS selector, matching UI elements in the current website view are highlighted with a blue border, as Figure 3.2 shows. Each time the user edits their code or moves their cursor to a different line, the highlights update to show the matching elements. This gives developers immediate feedback on which elements they are selecting and can help them identify mistakes.

3.3.1.2 UI snapshots

At runtime the tool captures UI "before" and "after" snapshots for each line of code, which the programmer can review to understand the effect of a given line. If the line has a CSS selector,

[2]A demo video for our prototype web automation IDE can be found at: https://dx.doi.org/10.7302/21954



```
[
    {
        "paramName1": "val1",
        "paramName2": "val2"
    }
]
```

```
// Write your script here
await page.goto("https://medium.com/topic/programming");
await page.waitFor(1000);

const authorElements = await page.$$(".bh.b.bj.aq a");
```

The Surface Laptop 4 Is a Great MacBook Alternative for the Microsoft-Curious
This year's refresh is focused on internals, so it's what's on the inside that counts
Owen Williams in Debugger
Apr 29 · 6 min read ★

30 Examples to Get You From a Novice to an Advanced Pandas User
A comprehensive and structured practical guide
Soner Yıldırım
May 2 · 11 min read ★

What I Learned at Work this Week: A Culture of Security and Reliability
This was a busy week at work, but one where I learned a lot more about the internal workings of our systems than…
Mike Diaz
May 2 · 8 min read

Coroutines for beginners: the only article you'll need to get started!
When we shifted to Kotlin at Gradeup, we realised that the www was pretty unfriendly for a coroutine beginner- there…
Faheem un nisa in gradeup
May 2 · 7 min read ★

3 Tools to Track and Visualize the Execution of your Python Code
5 min read

Why Every Developer Should Start Programming With C
4 min read

21 Best Practices for a Clean React Project
4 min read

Don't Start Coding With Python—Begin With C
5 min read

How to simulate a stock market with less than 10 lines of Python code
5 min read

```
0----------">…</a>
▼<div class="bg s">
  ▼<div class="bh b gp bj aq">
      <a rel="noopener" href="/@ow?source=topic_page--------------------
      ---20----------" class="blueBorder">Owen Williams</a> == $0
    ▶<a href="https://debugger.medium.com/?source=topic_page-----------
    ------------20----------" rel="noopener" class="blueBorder">…</a>
    </div>
  ▶<div class="n o">…</div>
```

Figure 3.2: It is challenging to select an author link on Medium because the `<a>` element does not have a semantic or specific selector. Instead, the parent `<div>` has a unique set of classes, so the programmer includes those in the selector – `.bh.b.bj.aq a`. Our prototype immediately highlights all matching elements on the page with a blue border, and lets the programmer see that they are mistakenly selecting not only author links but also publications.

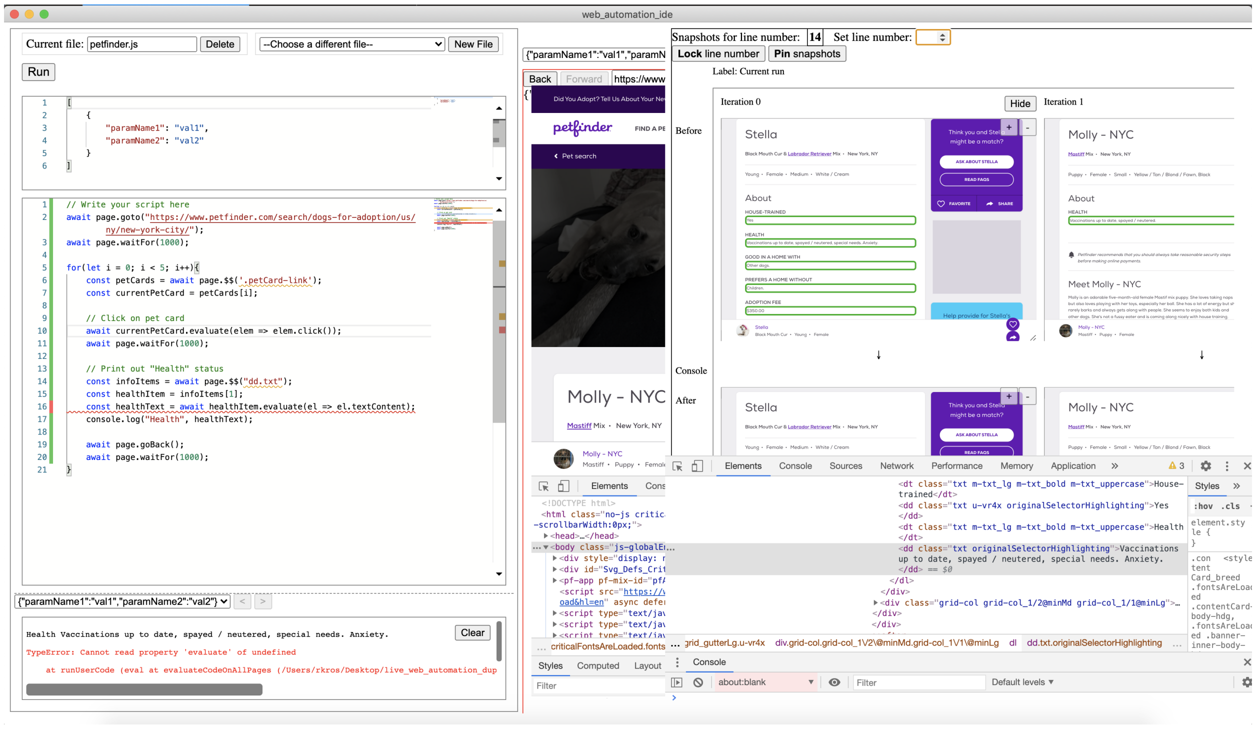
the matching UI elements are highlighted in the snapshots (Figure 3.3, green borders for elements matching line 14 selector `dd.txt`).

3.3.1.3 CSS selector validity feedback

An error message is provided and squiggle shown beneath each CSS selector string in the editor to indicate its validity in the context of the runtime page state: a yellow squiggle if the selector is found but not unique (Figure 3.3, lines 6 and 14) and a red squiggle if the selector is not found. Squiggles are updated live when the user edits a selector, with the selector checked against the UI "before" snapshot for that line. If snapshots are stale (i.e., earlier parts of the script have been edited since the last run), validity feedback is not shown for CSS selectors on that line.

3.3.1.4 Context and feedback across different runs

The prototype allows programmers to write scripts that contain loops and to run their script simultaneously across different sets of user inputs. This lets programmers test their code to make sure



```
1  [
2      {
3          "paramName1": "val1",
4          "paramName2": "val2"
5      }
6  ]
```

```
1  // Write your script here
2  await page.goto("https://www.petfinder.com/search/dogs-for-adoption/us/
       ny/new-york-city/");
3  await page.waitFor(1000);
4
5  for(let i = 0; i < 5; i++){
6      const petCards = await page.$$('.petCard-link');
7      const currentPetCard = petCards[i];
8
9      // Click on pet card
10     await currentPetCard.evaluate(elem => elem.click());
11     await page.waitFor(1000);
12
13     // Print out "Health" status
14     const infoItems = await page.$$("dd.txt");
15     const healthItem = infoItems[1];
16     const healthText = await healthItem.evaluate(el => el.textContent);
17     console.log("Health", healthText);
18
19     await page.goBack();
20     await page.waitFor(1000);
21 }
```

{"paramName1":"val1","paramName2":"val2"}

```
Health Vaccinations up to date, spayed / neutered, special needs. Anxiety.
TypeError: Cannot read property 'evaluate' of undefined
    at runUserCode (eval at evaluateCodeOnAllPages (/Users/rkros/Desktop/live_web_automation_dup
```

Figure 3.3: Our prototype lets users inspect UI snapshots per line of code, across execution contexts. Here, the script has failed in the `i=1` iteration of the loop, and the snapshots illustrate why. The UI snapshots for line 14 indicate that Stella (`i=0`) has five info elements (highlighted with a green border) matching selector `dd.txt` whereas Molly (`i=1`) only has one, which explains why the `infoItems[1]` indexing on line 15 failed for Molly's page.

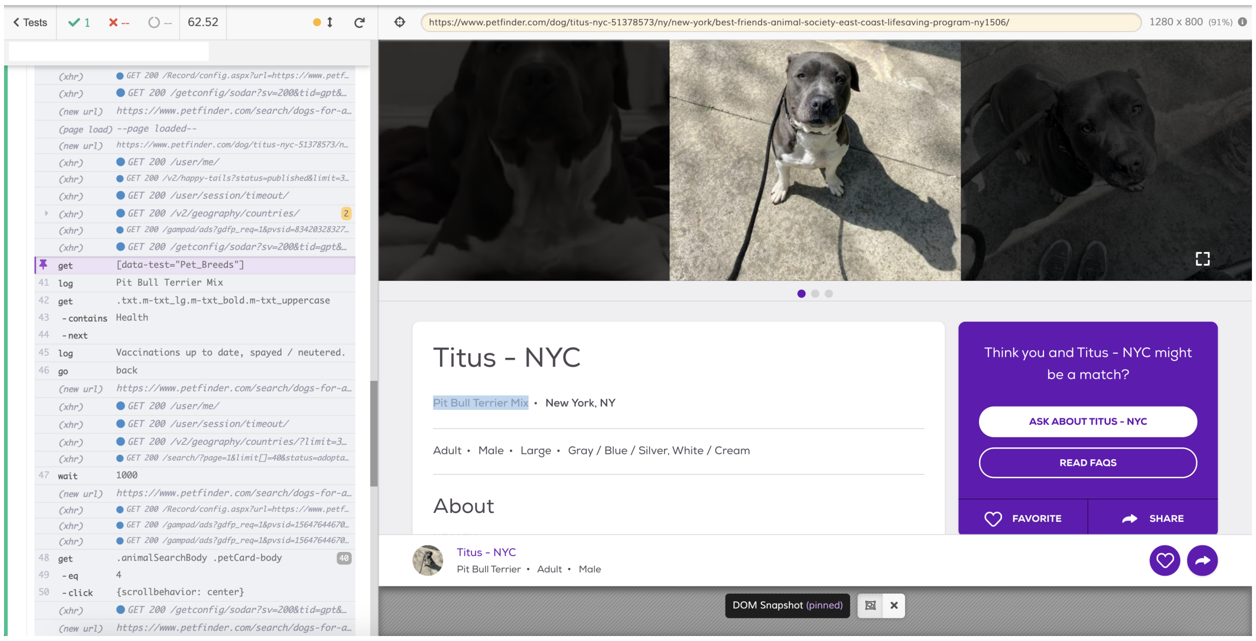
it works across scenarios or pages, and see corresponding UI snapshots and holistic CSS selector feedback for a given line of code in one place (Figure 3.3). This might help programmers discover that they have written a CSS selector or other logic that works in some scenarios or pages but not all.

3.3.1.5 Implementation

The prototype is implemented as an Electron [18] app, using Monaco editor [25] and Puppeteer [31] as the automation scripting library. It uses rrweb-snapshot [34] to capture and render UI snapshots of the DOM.

3.3.2 Cypress

Cypress [14] is an increasingly popular test automation environment that offers visual context and feedback about scripts at runtime. With Cypress, programmers write their code in a text editor and when they save their file, the results of their script are automatically updated in a web browser


Tests
1
--
--
62.52
https://www.petfinder.com/dog/titus-nyc-51378573/ny/new-york/best-friends-animal-society-east-coast-lifesaving-program-ny1506/
1280 x 800 (91%)
Titus - NYC
Pit Bull Terrier Mix • New York, NY
Adult • Male • Large • Gray / Blue / Silver, White / Cream
About
Think you and Titus - NYC might be a match?
ASK ABOUT TITUS - NYC
READ FAQS
FAVORITE
SHARE
Titus - NYC
Pit Bull Terrier • Adult • Male
DOM Snapshot (pinned)

```
    (xhr)        GET 200 /Record/config.aspx?url=https://www.petf…
    (xhr)        GET 200 /getconfig/sodar?sv=200&tid=gpt&…
    (new url)    https://www.petfinder.com/search/dogs-for-a…
    (page load)  --page loaded--
    (new url)    https://www.petfinder.com/dog/titus-nyc-51378573/n…
    (xhr)        GET 200 /user/me/
    (xhr)        GET 200 /v2/happy-tails?status=published&limit=3…
    (xhr)        GET 200 /user/session/timeout/
    (xhr)        GET 200 /v2/geography/countries/                  2
    (xhr)        GET 200 /gampad/ads?gdfp_req=1&pvsid=83420328327…
    (xhr)        GET 200 /getconfig/sodar?sv=200&tid=gpt&…
    get          [data-test="Pet_Breeds"]
41  log          Pit Bull Terrier Mix
42  get          .txt.m-txt_lg.m-txt_bold.m-txt_uppercase
43   -contains   Health
44   -next
45  log          Vaccinations up to date, spayed / neutered.
46  go           back
    (new url)    https://www.petfinder.com/search/dogs-for-a…
    (xhr)        GET 200 /user/me/
    (xhr)        GET 200 /user/session/timeout/
    (xhr)        GET 200 /v2/geography/countries/?limit=3…
    (xhr)        GET 200 /search/?page=1&limit[]=40&status=adopta…
47  wait         1000
    (new url)    https://www.petfinder.com/search/dogs-for-a…
    (xhr)        GET 200 /Record/config.aspx?url=https://www.petf…
    (xhr)        GET 200 /gampad/ads?gdfp_req=1&pvsid=15647644670…
    (xhr)        GET 200 /gampad/ads?gdfp_req=1&pvsid=15647644670…
48  get          .animalSearchBody .petCard-body                   40
49   -eq         4
50   -click      {scrollbehavior: center}
    (xhr)        GET 200 /getconfig/sodar?sv=200&tid=gpt&…
    (new url)    https://www.petfinder.com/search/dogs-for-a…
```

Figure 3.4: Cypress running a script that scrapes data from the Petfinder website. The user can hover or click on a particular command to see the UI state at that point in the execution, here item 40 where `[data-test="Pet_Breeds"]` is selected. The matched element ("Pit Bull Terrier Mix") is highlighted in the website view on the right.

augmented with Cypress UI panes (Figure 3.4). On the left, the Cypress command log presents the sequence of element selection and interaction commands the script executed. The programmer can hover or click on a given command (e.g., item 40 in Figure 3.4) to see the website's UI state at that point in the execution in the main browser viewport. For selection commands, the selected element(s) will also be highlighted in the website UI and the number of matched elements indicated on the command log item. Programmers can also use their browser's built-in developer tools as they normally would.

3.3.3 Study design

We recruited ten participants (eight male, two female; age 21–56, median age 29) from our university department, social media, and the Future of Coding community to participate in a 90 minute user study. We compensated participants with a $25 USD (or equivalent) Amazon gift card. Participants were all experienced programmers (eight with at least 5 years, two with 2–5 years experience) and all reported being comfortable working with CSS selectors and JavaScript methods for querying the DOM. Four participants reported some but not extensive experience with Cypress. Participants came from a variety of occupations (five professional developers, three PhD students, one undergraduate student, one CTO) and have varying experience with UI automation, ranging from none to more than five years. Each participant completed a web scraping task on each of the two conditions, our prototype and Cypress. The two web scraping tasks were:

1. *Medium*: Create a script that navigates to a Medium topic page[3] and for the first five articles, navigates to the article author's page, prints out the number of followers they have, and then navigates to their "About" page.

2. *Petfinder*: Create a script that navigates to a Petfinder search results page[4] and for the first five dogs, navigates to the dog's page, prints out the dog's breed, and prints out information about the dog's health.

We chose these two websites because they are non-trivial to write scripts for: many elements do not have IDs, classes, or attributes that are semantically meaningful to select by; and there are differences across pages on a given site, either in the content shown or DOM implementation.

We counterbalanced task order and website/tool pairings. Participants were given 25 to 30 minutes per task, with the exception of P1 who was only given 22 minutes for the Cypress task. Before each task, participants watched an eight minute tutorial video about the tool that illustrated how it works and its different UI context and feedback features. We gave them a reference sheet and allowed them to search the web for resources during the task. Due to short task time and to help fill knowledge gaps, during the task we answered questions they had about Cypress, Puppeteer, and CSS syntax and provided hints if the participant was stuck for awhile. After completing both tasks, we conducted a brief interview.

3.3.4 Results

Participants found aspects of both tools useful, in particular the feedback on which UI elements are being selected. We first give an overview of the main challenges participants encountered in writing generalized scripts. We then discuss the kinds of context and feedback participants needed, in what ways the tools provided them, and participants' opinions on specific UI context and feedback features.

3.3.4.1 Challenges

A primary challenge of the tasks was identifying selector logic that generalizes appropriately. Specifically, some of the common challenges were:

Selecting content correctly across pages when it has no semantically meaningful class names and content order varies. Petfinder dog profile pages (e.g., as seen in the snapshots in

[3]https://medium.com/topic/programming
[4]https://www.petfinder.com/search/dogs-for-adoption/us/ny/new-york-city/

Figure 3.3) include information about the dog's health, friendliness, adoption fee, and more, but the exact categories and number of categories shown per dog varies. This information is presented in DOM elements that have no semantically meaningful class or attribute names, making them more challenging to extract. When we asked participants to scrape dogs' health information across pages, six participants tried selecting by a general selector like `dl dd.txt`, which selects text from all information categories, and then indexing into the results list to choose the second item (which, on the first dog's page, corresponds to the health information). A couple participants noted that this might not work, but tried it anyway. Once they ran their script, they got an error and saw that the second dog only has "health" information and no other categories, so their indexing approach is not robust (Figure 3.3). Three other participants up front chose to select by text value, which was a successful approach – they first selected the element containing the text "Health", then chose its next sibling to retrieve the information itself.

Selecting an element correctly when it has different CSS class names across different pages. The Medium website uses obscure CSS class names that vary across pages. All authors have a "number of followers" element in their page header but the CSS classes are different per author. Many participants constructed their selector for the "number of followers" element using the CSS classes listed on the first author's page (e.g., `.cd.gg.t a`), which then did not work on other author pages. Four participants encountered this problem only after they ran their script and saw a "no elements matched" error. However, two other participants avoided this problem by instead using the more robust and semantic selector `[href$="/followers"]`, selecting elements whose `href` attribute ends in `/followers`. One participant made this choice based on intuition, and the other first chose to review multiple author pages' to compare their DOM trees and check if the class-based selector they chose would generalize, and they saw it did not.

Identifying elements that are clickable. For subtasks that involved clicking, participants were easily able to identify the correct visual element on the page to click on. However, when constructing a CSS selector, some participants selected an ancestor element of the clickable <a> element (e.g., because the ancestor has more specific classes or attributes), but the ancestor in some cases actually does not respond to click events. For example, when selecting an author link from the Medium starting page, participants discovered that author links do not have semantically meaningful or specific classes or attributes (Figure 3.2). The best option is to use the author link's parent's class names (e.g., `.bh.b.bj.aq`) as part of the selector. Five participants used a selector like `.bh.b.bj.aq` but forgot to further query to select the actual <a> element, and were therefore confused when clicking `.bh.b.bj.aq` did not navigate to the author's page. Two participants experienced the same problem on Petfinder.

Choosing a selector that is specific enough to select certain elements but not others. As mentioned above, participants used selectors like `.bh.b.bj.aq` and then further queried by `a` to select author links from the Medium website. Many participants therefore simply chose `.bh.b.bj.aq a`, which selects all `<a>` with an ancestor that has classes `.bh.b.bj.aq`. This selects the author link (e.g., text "Owen Williams" in Figure 3.2), but also incorrectly selects publication links (e.g., text "in Debugger"). Three participants used this selector and only discovered it was too general once they ran their script and saw it navigating not only to author pages but also publication pages. On the other hand, a different participant (P2) realized his selector was too general before he even ran his script, by taking advantage of our prototype's dynamic element highlighting feature. For his first selector attempt `.bh.b.bj.aq a`, author and publication links were highlighted (Figure 3.2), which is not what he wanted. He then adjusted his selector to be `.bh.b.bj.aq a:first-child` and saw the blue highlighting update to highlight only the author links as he wanted.

3.3.4.2 Element selection context and feedback

Participants appreciated receiving feedback and UI context for the elements their script selected. Seven of ten participants verbally expressed that they found the CSS selector inline feedback squiggles in our prototype useful. Participant P6 said *"I found that really useful, the inline contextual help on that, because that helped me like immediately identify, 'OK, it was running this line, it couldn't find this thing' "*, referring to a CSS selector he wrote that had a typo. A couple participants also noticed that the selector feedback squiggles update when they edit a selector string – *"I'm seeing that when it doesn't match anything, that turns red. If I had known it existed at the beginning I would've used that instead of fiddling around in the console. That feedback is really nice"* (P7). Participants similarly appreciated Cypress's runtime element selection feedback.

Participants also appreciated selected elements being highlighted in the website view and UI snapshots, and used this to identify if they selected the desired elements and make corrections accordingly. For example, with Cypress, participant P2 realized that simply selecting by the text "Health" on the Petfinder website was not specific enough to query the "Health" category, because he saw another instance of "Health" on the page was getting selected instead. With Cypress, P9 realized that her selector for selecting author links on Medium was actually incorrectly selecting publication links some of the time. Seven participants said the dynamic element highlighting our prototype offers in the main website view is useful, commenting on how the dynamic highlighting shortens the feedback loop and provides an easy way for checking if their selector is selecting the right elements. P2 in particular used the dynamic highlighting feature heavily, iteratively writing and adjusting his selector based on the highlighting feedback, and for example realizing he was incorrectly selecting publication links as discussed above in the "Challenges" section and Figure 3.2.

Participants mentioned additional kinds of live feedback they would like to see. P1 and P6 want to see live UI snapshots that update immediately each time the user edits their code. P2 also wants to see variable and element attribute values evaluated live – *"I would probably like to see what the [hrefs] capture, because I usually spend a lot of time debugging...like what would be evaluated...a bit of like a REPL experience like in dev tools or console"*.

3.3.4.3 Understanding page states

In creating their scripts, participants needed to understand the pages with which their script was interacting. When participants needed to confirm that their script commands worked as intended and navigated to the correct sequence of pages, most participants simply watched their script run in the main website view. One participant (P4) used Cypress's snapshots heavily for understanding unexpected page navigation. She was confused why a certain author was visited twice and used Cypress's snapshots to discover that the order of authors listed on Medium had changed during the course of her script's execution, which was using the live website content. To write generalized element selection logic, participants needed to understand the similarities and differences between different author pages on Medium and different dog pages on Petfinder, and to do this they manually navigated in the main website viewport.

Although participants did not heavily use UI snapshots, several participants commented on how they could be useful. P8 commented on how being able to compare different pages is important – *"I realized that each page might be different, I wondered if that selector from the last page is going to be generalizable...I wonder if there's like a better way than me just manually clicking through [the pages], I was imagining if there's a visual comparison, where I got to select multiple sites at once..."*. We suspect UI snapshots were underutilized by participants because 1) the short task time was not enough to become fully familiar with UI snapshots and internalize where they would be useful and 2) UI snapshots might be a tool that is appropriate for less frequent situations.

3.3.4.4 Traditional debugging approaches

Even with the various UI context and feedback features available, most participants still leveraged traditional web UI development and debugging techniques. Seven participants executed selector query commands in the browser dev console to experiment with candidate selectors, check which element(s) they match, and further inspect these elements.

3.4 Design Implications

As our study results show, writing web automation code presents a unique set of challenges and information needs. We can divide the information needs of web automation developers into roughly two categories: *context* and *feedback*. In this section, we describe our recommendations for the kinds of contextual information and feedback that future web automation tools should provide.

3.4.1 Contextual Information

Web automation code references the internal structure of the web page on which it runs. Providing the right context about the target web page can make it easier for developers to write Web automation macros by bridging the "gulf of execution" [102].

- *DOM nodes and values*: The code editor should provide inline access to the values of variables, selected elements, and their attributes, perhaps available on hover. Inline access is important for helping developers early on understand the elements they are selecting and the values their script is producing.
- *UI snapshots*: UI snapshots of each step of the execution should be provided to help programmers understand whether they are selecting the correct elements, whether the expected behavior occurred, and what the page state is after a given command. Many participants in our second study found this helpful.

3.4.2 Effective Feedback

Immediate feedback can help developers discover problems in their code early by bridging the "gulf of evaluation" [102]. It may be technically challenging to provide live feedback, as web automation scripts do not run immediately but rather only as quickly as web pages navigate and render. For efficiency, a live feedback tool might keep a copy of the page state per line of code, so that whatever line of code the programmer edits next, the script can be run starting from that particular line.

- *Feedback on selectors*: The code editor should provide inline feedback per element selection command, indicating clearly the number of matching elements and whether any elements are hidden. The exact elements that are selected should be highlighted in a UI snapshot of the corresponding page state. Many participants in our second study found this feedback helpful. UI snapshots should be shown in the periphery of the editor so the programmer can validate their element selection logic as they write it.
- *Feedback on interactivity of elements*: The code editor should give feedback on whether the selected element can be interacted with as the developer intends. For example, if the programmer tries typing into an element that cannot be typed into, clicking on an element

that cannot be clicked, or setting the `value` for an element that has no `value` attribute, the editor should show an error rather than letting the command silently fail. This feedback is important because information about element event handlers is not always clearly visible in browser dev tools. None of the environments we evaluated provide feedback on whether elements can be interacted with as intended, and as a result a few of our participants were puzzled that their interaction commands did nothing.

- *Feedback across pages*: Many participants in our studies wrote element selection logic that worked for one website page but that they later realized did not work for others. Perhaps web automation tools should proactively suggest or prompt users to identify multiple pages that the script should run correctly on. This could help programmers earlier on understand differences across pages and write code that appropriately generalizes.
- *Longitudinal feedback*: Developers cannot anticipate and have no control over how and in what ways third-party websites will change over time. We saw that the DOM for websites from our first study changed within the course of a year, and some participant-chosen selectors would not have worked on those other website versions. It would be valuable for web automation tools to help developers identify when a website has changed in ways that will break their script or cause its behavior to change, and to help developers repair their script accordingly.

3.5 Discussion and Limitations

Our study design allows us to present a qualitative description of the challenges and needs of programmers writing web automation scripts. However, due to its small participant size and exploratory nature it is lacking quantitative measures of how long certain task types take and whether UI context and feedback features offer speedup. Additional studies with more participants, a broader set of tasks, and longer study sessions would be informative.

UI context and feedback features will *inform* programmers as they develop and debug, but will not actually *write* the code for them. Recent innovations in programming-by-demonstration [55, 88] could help generate automation scripts, but for full control, programmers will still need to reason about and choose navigation logic and CSS selectors themselves.

3.6 Conclusion

Programmers writing web automation scripts have specialized needs, as they need to interpret third-party websites and programmatically mimic user interactions. Through two user studies, we found that these programmers need contextual information about the UIs they are interacting with and feedback on their element selection and interaction code. We hope our research can help guide the design of future web automation tools. We also believe many of our design implications may be relevant for UI test automation and UI programming.

CHAPTER 4

Creating UI Automation Leveraging End-User Natural Language Parameterization[1]

4.1 Introduction

The Web is a rich source of information. Web automation makes it possible to programmatically access this information by mimicking user interactions, such as clicking on buttons and typing text into fields, on a web page. This can be beneficial in a variety of scenarios. For example, enabling voice-based access [38] to web content could make it more accessible, and macros could allow users to complete tasks that would be tedious when performed manually. However, the time, expertise, and effort required to write automation code makes it impractical to support the long tail of user needs.

Prior research has shown that Programming-By-Demonstration (PBD) [93, 62] is an effective way to allow users—including users without programming experience—to create user interface (UI) automation macros [88, 90, 91, 110, 104]. The user demonstrates how to perform the task that they want automated, and then the PBD system generates code capable of mimicking the user's actions on a UI. However, a challenge of PBD systems is inferring how to *generalize* from one demonstration—inferring a domain of similar tasks and performing tasks within that domain. In this paper, we focus on improving *parameterization* of PBD-generated automation macros, in the context of natural language (NL)—specifying the slots in NL queries and what values they might have. Parameterization is a key method for scaling the domain of tasks that automation can handle.

We propose leveraging end-users to identify macro parameters and values that match their intent. We designed a novel PBD system, ParamMacros, that allows end-users to create custom macros that answer parameter-based questions about website content. End-users start with a con-

[1]This chapter is adapted from the publication: Rebecca Krosnick and Steve Oney. ParamMacros: Creating UI Automation Leveraging End-User Natural Language Parameterization. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2022).*

crete natural language question they have, then through a text annotation interface identify parts of their question that could change (i.e., parameters) and provide possible alternative values. Using this parameterized natural language question, the end-user now selects a question instance (i.e., a value per parameter) and demonstrates on the website the correct answer for that question and the necessary page interactions to find that answer. ParamMacros then infers a generalized program based on the user-provided parameters and demonstration.

PBD systems Sugilite [88] and Appinite [90] also enable end-users to create custom automation that supports their specific natural language requests. To identify related UI elements during program inference, Sugilite primarily considers sibling nodes, and Appinite uses its natural language understanding (NLU) to interpret user NL and accordingly identify relevant relationships from its UI knowledge graph. A key difference in our system ParamMacros is that it leverages user-provided parameters and values to identify relevant patterns as it traverses the Document Object Model (DOM) [16] hierarchy during program inference. Complex relationships exist between elements at many levels in a UI hierarchy, and we offer a new approach to identifying those relationships.

We focus on website content that has semantic entries and attributes (e.g., a list of movies and their metadata, a table of sports statistics). Through a user study we show that users can identify meaningful parameters and effectively create demonstrations, and that users think creating such generalized automation macros would be useful.

We contribute the following:

- The idea of having end-users identify parameters in their natural language questions as input to PBD systems.
- A text annotation interface for identifying parameters and alternative values.
- ParamMacros, a PBD system for creating automation macros that answer parameterized questions about website content.
- An inference approach that leverages structural patterns in the website DOM to identify candidate parameter values.
- A user study suggesting the feasibility and usefulness of users generalizing their own natural language requests.

4.2 System Usage Scenario

ParamMacros enables end-users to create custom parameterized macros for answering questions about content on websites. In this section, we will use an example to illustrate the process of

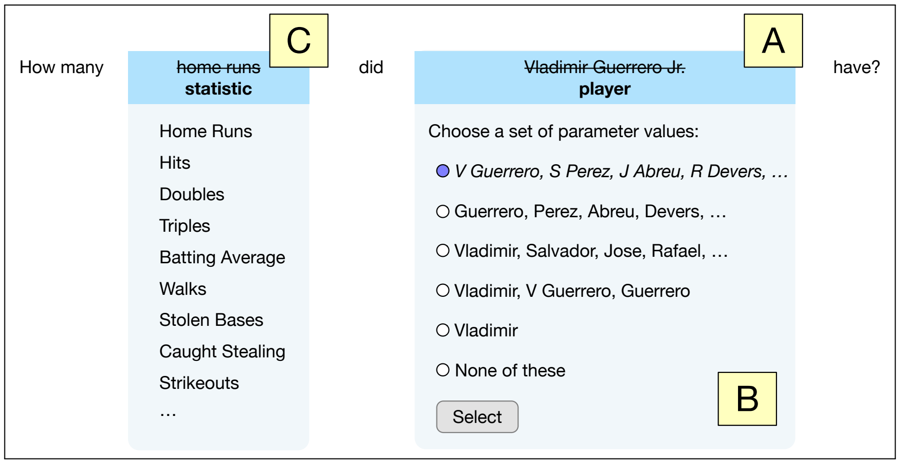

How many
~~home runs~~
statistic
C
did
~~Vladimir Guerrero Jr.~~
player
A
have?
Home Runs
Hits
Doubles
Triples
Batting Average
Walks
Stolen Bases
Caught Stealing
Strikeouts
...
Choose a set of parameter values:
● V Guerrero, S Perez, J Abreu, R Devers, ...
○ Guerrero, Perez, Abreu, Devers, ...
○ Vladimir, Salvador, Jose, Rafael, ...
○ Vladimir, V Guerrero, Guerrero
○ Vladimir
○ None of these
Select
B

Figure 4.1: An illustration of ParamMacros's UI for parameterizing natural language queries. The user has chosen to (A) generalize "Vladimir Guerrero Jr." to make the parameter *player* and (C) generalize "home runs" to parameter *statistic*. The system proposes possible alternative values (B) for each parameter for the user to select from.

creating such macros. The process consists of two steps for end-users: 1) identifying the pieces of a concrete question that can generalize and expressing these through *parameters* and *alternative values*, and 2) creating the automation macro through programming-by-demonstration, by giving an example of the correct answer for a particular set of parameter values.

Alice is a baseball fan and frequently asks questions about player statistics, for example, "How many home runs did Vladimir Guerrero Jr. have?", "What was the most triples anyone had?", and "For the player who had the most stolen bases, how many walks did they have?". She decides to use ParamMacros[2] to create automation macros to answer these kinds of questions from data on the Major League Baseball (MLB) statistics web page[3][4].

4.2.1 Generalizing a question

Alice starts by creating an automation macro to answer a specific question: "How many home runs did Vladimir Guerrero Jr. have?". She knows she might want to ask similar questions in the future about other players, too. She expresses this question variation in the system interface by highlighting "Vladimir Guerrero Jr." with her cursor to create a **parameter** (Figure 4.1A). This parameter (which she names *"player"*) replaces "Vladimir Guerrero Jr." and serves as a slot to represent any MLB player's name. She now needs to express the possible MLB player names.

[2]A demo video for ParamMacros can be found at: `https://dx.doi.org/10.7302/21953`

[3]Using a replica of `https://web.archive.org/web/20220201043626/https://www.mlb.com/stats/`

[4]Although data in this scenario is tabular, our system also works with websites containing other kinds of hierarchically structured data.

ParamMacros proposes potential **parameter values** (Figure 4.1B), which it extracted from the MLB website. Alice reviews the different options, sees that the first two radio buttons list the player names she was expecting, and chooses the first one (e.g., V Guerrero, S Perez, J Abreu). This identifies the possible values for the *player* parameter.

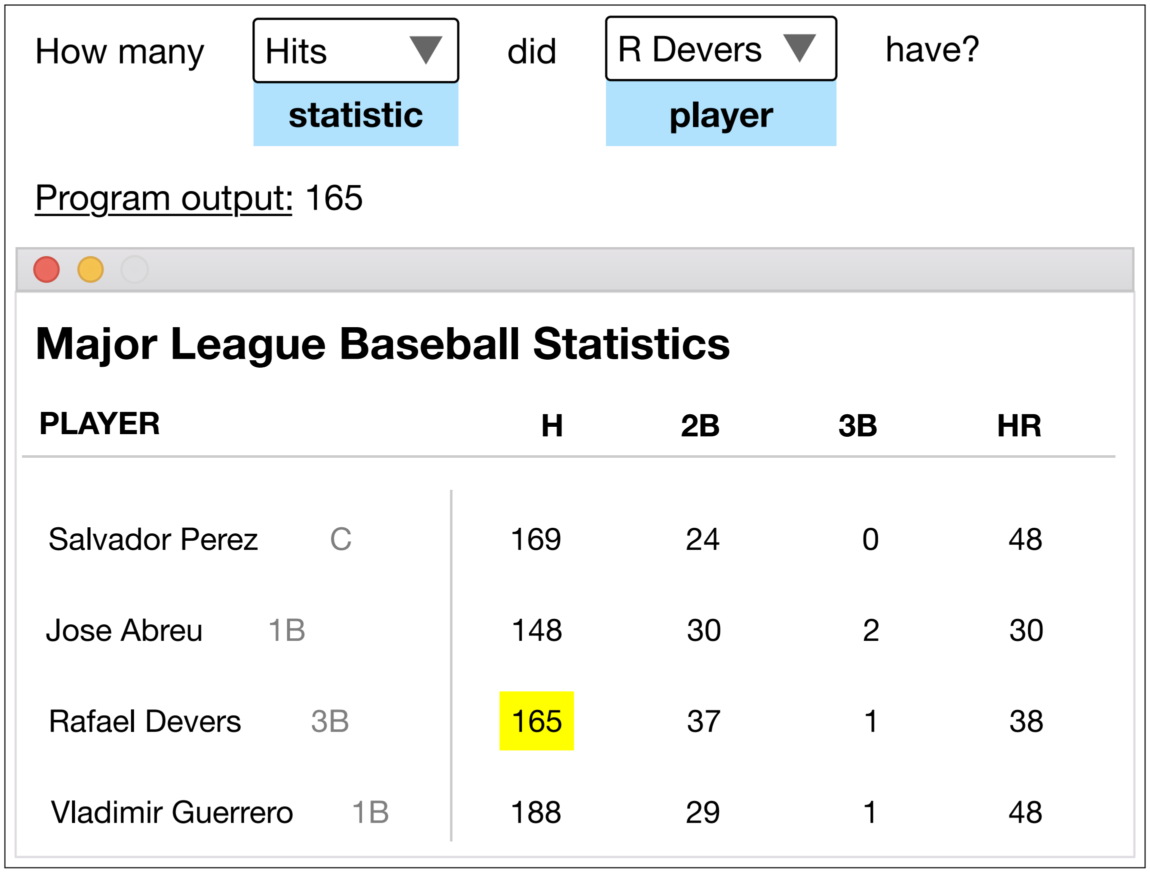
Alice knows that she also might want to ask this kind of question not only about home runs, but about any baseball statistic. She therefore also parameterizes "home runs" to a parameter named *statistic* and selects an appropriate auto-extracted parameter value list (e.g., "Home Runs", "Hits", "Doubles") (Figure 4.1C). Alice now has the generalized question "How many <*statistic*> did <*player*> have?" that represents all the questions she might ask about any statistic for any player.

4.2.2 Creating an automation macro

Alice can now create an automation macro for her generalized question. To do this, Alice needs to provide a **demonstration** of how to answer a particular instance of the question. ParamMacros's inference engine will then infer a generalized automation macro from that single demonstration, through a process described later in this paper (section 4.3). Alice demonstrates how to answer her original question "How many *Home Runs* did *V Guerrero* have?" through ParamMacros's demonstration interface (not shown, but similar to the execution interface in Figure 4.2). She provides the context for the demonstration by selecting *Home Runs* from the <*statistic*> parameter dropdown menu and *V Guerrero* from the <*player*> parameter dropdown menu. She then clicks the "Start recording" button. Now she searches the page for the correct answer (the "HR"—short for Home Runs[5]—column value for Vladimir Guerrero), selects the text (48—the correct value), and clicks "Extract". She stops recording the demonstration and ParamMacros generates the macro.

Alice now tests the macro to make sure it behaves as she intended. She starts by running the macro with the parameter values <*statistic*>=*Home Runs* and <*player*>=*V Guerrero* that she used in her demonstration and sees that the output, 48, is correct. She also sees that the macro highlighted the answer location on the page in yellow. She then tries running the macro on different sets of parameter values to make sure it generalized correctly. For example, she runs the macro using <*statistic*>=*Hits* and <*player*>=*R Devers* and is pleased to see that the macro returns the correct answer, 165 (the "H" column value for Rafael Devers) (Figure 4.2).

[5]Our inference algorithm discovers that "HR" corresponds to "Home Runs" because the "HR" UI element contains a visually hidden UI element with the text "Home Runs".



Program output: 165

Major League Baseball Statistics

PLAYER		H	2B	3B	HR
Salvador Perez	C	169	24	0	48
Jose Abreu	1B	148	30	2	30
Rafael Devers	3B	165	37	1	38
Vladimir Guerrero	1B	188	29	1	48

Figure 4.2: An illustration of ParamMacros's execution interface and the Major League Baseball statistics website. When the user runs the generated macro with the inputs *<statistic>=Hits* and *<player>=R Devers*, it returns and highlights the correct answer, 165.

4.2.3 Program description

Although the inferences in the above example were correct, it is important to consider how users can recover from incorrect inferences. ParamMacros supports this through an interface that represents a high-level description for each macro. Each description explains the logic for which element is selected for each program step, and whether it depends on any parameter values. For example, the program description for "For the player who had the *<most/least>* *<stat1>*, what was their *<stat2>*?" (Figure 4.3A), explains that the entry (e.g., row) to select is determined by the entry whose *<stat1>* parameter value is the *<most/least>*, and that the *<stat2>* parameter specifies which attribute (e.g., column) value to print out. We show a comparable kind description for "filter" rules, where the entry to select is determined by a particular parameter.

Radio buttons show alternative selection rules (e.g., in Figure 4.3A to ignore the *<stat2>* parameter and always just print out from the *Batting Average* column) if Alice believes the default logic is wrong. The ability to adjust selection rules could be useful if there were ambiguity in

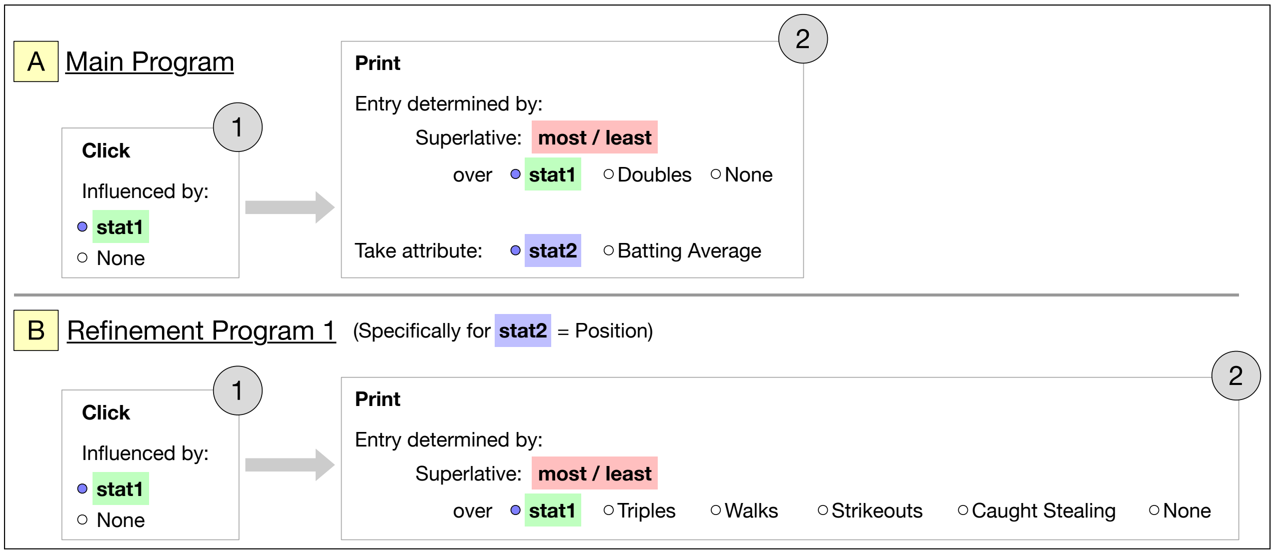

A
Main Program
1
Click
Influenced by:
stat1
None
2
Print
Entry determined by:
Superlative: most / least
over stat1 Doubles None
Take attribute: stat2 Batting Average
B
Refinement Program 1 (Specifically for stat2 = Position)
1
Click
Influenced by:
stat1
None
2
Print
Entry determined by:
Superlative: most / least
over stat1 Triples Walks Strikeouts Caught Stealing None

Figure 4.3: Program description for "For the player who had the <*most/least*> <*stat1*>, what was their <*stat2*>?" The program (1) first clicks a header in the statistics table to sort the data, and then (2) prints out a value from the table. (A) General program logic used for all parameter input values except <*stat2*> = *Position*. (B) Logic generated from the user's refinement demonstration; used only when the user runs the program with <*stat2*> = *Position*.

the demonstration (e.g., if Alice had selected "Hits" for both <*stat1*> and <*stat2*>, the inference engine would not know if the value to print out should be <*stat1*> or <*stat2*>).

4.2.4 Refining an automation macro to support edge cases

As Alice creates her macro to answer the query "For the player who had the <*most/least*> <*stat1*>, what was their <*stat2*>?", she decides that in addition to the list of auto-extracted statistics for <*stat2*> (e.g., Home Runs and Strikeouts), she would also like to ask about the player's "position" (i.e., their role on the team, such as pitcher, second base, outfield). However, when she runs her macro, she realizes it only returns the correct answer for the original statistic values and not for <*stat2* = *position*> (the word "position" does not appear as text on the page, so our algorithm does not know where to find the answer; explained more in section 4.3).

To work around this problem, Alice creates a **refinement demonstration** to create entirely separate program logic specifically for when the parameter <*stat2*> equals "position". Alice first specifies the single parameter and value pair that she wants to create the refinement demonstration for when <*stat2* = *position*>. She then records the demonstration, using the same process as she has in the past. The updated macro is now comprised of two subprograms (Figure 4.3). Now when Alice runs the macro, it will run "Refinement Program 1" if Alice has set <*stat2*> to "position"; otherwise it will run the original "Main Program". The macro now correctly outputs the position

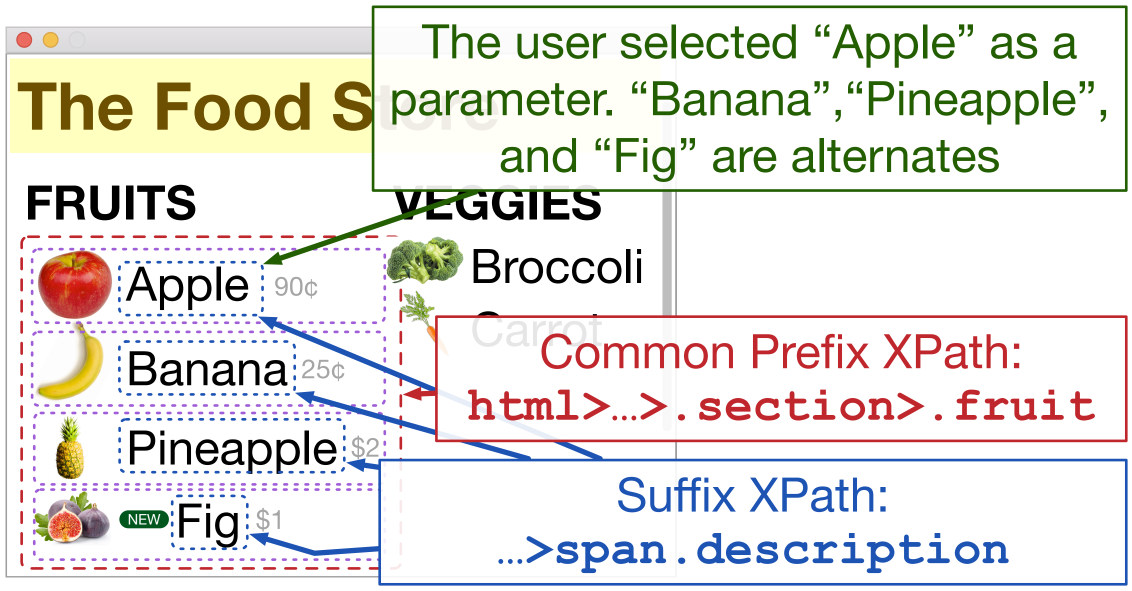

The Food Store
FRUITS
VEGGIES
Apple 90¢
Banana 25¢
Pineapple $2
NEW Fig $1
Broccoli
Carrot
The user selected “Apple” as a parameter. “Banana”,“Pineapple”, and “Fig” are alternates
Common Prefix XPath: html>…>.section>.fruit
Suffix XPath: …>span.description

Figure 4.4: An explanatory illustration of our inference algorithm on an imaginary website titled “The Food Store”. Here, the user has selected “Apple” as a parameter in their NL query and wants “Banana”, “Pineapple”, and “Fig” to be alternative values. Our algorithm infers a common suffix across candidate parameter values and a common suffix across target elements.

for questions of the form “For the player who had the <*most/least*> <*stat1*>, what was their <*stat2 = position*>?”.

4.3 Inference Algorithm

ParamMacros’s inference algorithm takes advantage of common patterns in the Document Object Model (DOM)—a tree that represents the webpage content. ParamMacros identifies potential parameter values within the website DOM and infers how users’ actions may generalize to new parameter values.

4.3.1 Parameter values

4.3.1.1 Proposing candidate parameter values

When the user selects text from their question to parameterize, ParamMacros tries to identify other possible values for this parameter. Our algorithm first uses fuzzy string matching to find the best on-page match for the selected text above a minimum threshold.

If an on-page match for the user’s sample parameter value is found, ParamMacros begins to search for other possible parameter values. For example, if the user asks “How much does one

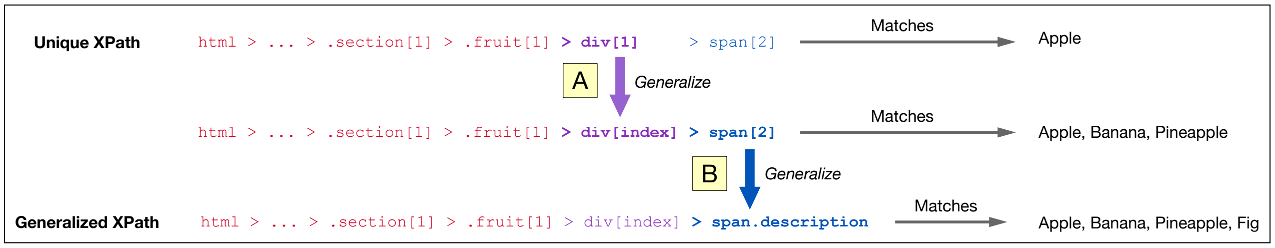

Unique XPath
html > ... > .section[1] > .fruit[1] > div[1] > span[2]
Matches
Apple
A
Generalize
html > ... > .section[1] > .fruit[1] > div[index] > span[2]
Matches
Apple, Banana, Pineapple
B
Generalize
Generalized XPath
html > ... > .section[1] > .fruit[1] > div[index] > span.description
Matches
Apple, Banana, Pineapple, Fig

Figure 4.5: The process to transform a single value's XPath to a generalized XPath formula that works across parameter values. The algorithm starts with a unique XPath matching the original parameter value, "Apple". (A) The algorithm then identifies possible "iteration points" that generate alternative parameter values; here we show one possible iteration point, which generalizes the specific node `div[1]` to `div[index]`, resulting in the XPath formula now also matching "Banana" and "Pineapple". (B) The algorithm then tries to make each XPath node more robust, opting for more semantically meaningful selectors. Here, the algorithm generalizes `> span[2]` to `> span.description`, resulting in the XPath formula now also matching "Fig".

<Apple> cost?" on the page in Figure 4.4 and selected "Apple" as a parameter, they might want the algorithm to infer that "Banana", "Pineapple", and "Fig" are alternative values. Our algorithm first builds an XPath[6][7] query that uniquely matches the element. It builds an index-based XPath (e.g., not classes alone) since this is the easiest way to ensure a unique XPath. For example, in Figure 4.4, a unique path for "Apple" might be `html >...> .section[1] > .fruit[1] > div[1] > span[2]`. A key insight of our algorithm is that other candidate values often have *similar* paths. Replacing `div[1]` with `div[2]` in the above XPath would yield the text element for "Banana" (and `div[3]` yields "Pineapple").

We refer to the first portion of the query (colored red from `html` to `.fruit[1]`) as the "common prefix". It represents the path to the element that contains the list of items. The second portion (colored purple: `div[1]`) points to the specific element that contains the text "Apple", the image of the apple, and the price. We refer to this as the "iteration point". The last portion (colored blue: `> span[2]`) points to the portion of that specific element with the "Apple" text (to exclude the image and any other irrelevant elements). We refer to this as the "common suffix".

Our algorithm iteratively determines the common prefix, iteration point, and common suffix. First, the initial XPath query it generates uses indices to identify unique elements (as we describe in the next subsection, some of these will be replaced with more robust class queries). Next, it tries to identify an ideal iteration point (Figure 4.5A). There are many possible iteration points for

[6]XPath is a language for querying the DOM based on HTML attributes and hierarchy; `https://en.wikipedia.org/wiki/XPath`

[7]For the sake of brevity, we use a CSS query syntax in this paper rather than XPath (which our system uses). In this syntax, `body > div[3] > .cl1 > span.cl2` matches an element with the tag `span` and class `cl2` that is a direct child of an element with class `cl1` that in turn is a direct child of the third `div` (index `3`) inside a `body` element.

a given XPath query. In the above query for Figure 4.4, placing the iteration point at `.fruit`, for example, might yield "Apple" and "Broccoli" (the first children of similar-looking elements) as possible values. To disambiguate, our algorithm first iterates through all possible iteration points and ranks them by number of valid results (whether the common suffix leads to a text node). We ask the user to make the final decision about which candidate values to use (if any) since it often is impossible to accurately infer the user's intent.

Once the user selects one of the proposed parameter values lists (or manually writes values), if the user edits or adds any values, the algorithm goes through a similar process to identify the parameter values' locations (i.e., XPaths) on the page. It is important to know the parameter values' locations on the page because later on, our program inference algorithm leverages parameter values' locations for identifying which parameters a given demonstration event might depend on, if any.

4.3.2 Generalizing parameter value XPaths

Now that we have attempted to find XPaths for all of the parameter values, the algorithm now tries to generalize these XPaths to have a common XPath suffix (Figure 4.5B). This is important because later on the inference algorithm relies on parameter value XPaths having the same suffix when it creates generalized rules. Parameter values that visually look similar will not necessarily have the same XPath suffix initially. In the example from Figures 4.4– 4.5, the first step of our inference algorithm produced `> span[2]` as the XPath suffix, to select the second child of the parent element (as the fruit images are the first child of each). This would match "Apple", "Banana", and "Pineapple". It would **not** match "Fig", however, because the "Fig" text is the **third** child instead of the second child (the 'NEW' badge is the second child).

We want to create automation macros that are robust to these kinds of DOM variations. To create a generalized XPath suffix that matches as many parameter values as possible, we traverse through the generated XPath one level at a time and try to find a common class or attribute name across parameter values to replace that XPath node with. Classes and attributes are likely more semantically meaningful than the default index-based XPath and are robust to index offsets. For the example in Figure 4.4, our algorithm would therefore find the more general suffix `> span.description` (Figure 4.5B).

4.3.3 Inferring parameter-based automation logic

The algorithm then tries to infer which parameters (if any) the user might want their program to depend on. It does this by looking for correspondences between the user's demonstration events and the XPaths of the parameter values the user selected by leveraging two techniques, described below.

4.3.3.1 Inferring row/column-based selection

For a given demonstration event, the algorithm tries to identify if the target element is within a table (either an HTML `table` or a `div`-based table). The algorithm tries to identify semantically similar siblings (i.e., potential rows and columns) by traversing up through the DOM hierarchy and at each level computing the children nodes' similarity with each other, using Dice's coefficient to measure the string similarity of the nodes' `outerHTML` (i.e., the node and its full subtree). We then use the two DOM levels with the highest similarity scores and consider these as our rows and columns (we discuss limitations of this approach in section 4.3.4), and identify where the target element falls within these rows/columns.

Now the algorithm can try to infer if the target element's row and/or column could be based on the specified parameter/value pairs. For identifying whether the selected target element *column* could correspond to a parameter, we essentially try to determine if the table's columns correspond to a particular parameter's set of values by trying to align columns with parameter value elements. Once we identify which parameter *p*'s values (if any) the table's columns correspond to, we now check if the value the user assigned to parameter *p* for this demonstration matches the target element's column's parameter value. If these align, then we infer that the target element's column is determined by parameter *p*.

The algorithm relatedly uses its knowledge about the table and selected parameter/value pairs to infer the reason that the target element's *row* was selected. It checks to see 1) if a selected parameter value appears as text in the row, acting as a **filter** for the row (e.g., filtering by the *player* name) and 2) if the selected row satisfies a **superlative** for one of its columns (e.g., row with the highest number of Home Runs).

4.3.3.2 Inferring entry-based selection

If the algorithm cannot find a meaningful row/column pattern, it tries to determine if the target element is an "attribute" associated with a specific parameter value. In Figure 4.4, if the user asks "What is the price of <*fruit*>?" and demonstrates the answer "$2" for <*fruit* = *Pineapple*>, the algorithm infers that "$2" was printed because it was "closer" to "Pineapple" than to any of the other fruit values, i.e., because $2 and Pineapple have the same parent, whereas $2 and the other fruits only share the grandparent `html >...> .section > .fruit`.

Our algorithm then identifies the relative XPath relationship between the parameter value and the target element so it can form a general rule to apply for other parameter values in the future.

For example, here, the XPath suffix for the "Pineapple" text is `> span.description` and the XPath suffix for Pineapple's price ($2) is `> span[3]`. The inferred rule would be to get the XPath for the input parameter value (e.g., Apple, Banana) and replace `span.description` with `span[3]` to find the new target element (the price) to return to the user.

At this point, this inferred rule will work if the macro is run with <*fruit*> set to "Apple", "Banana", or "Pineapple", but will return the wrong answer when run for "Fig". This is because the suffix for Apple, Banana, and Pineapple's price is `> span[3]` but the suffix for Fig's price is `> span[4]` (because of the offset due to the 'NEW' badge). Therefore, the XPath the macro infers for Fig's price would erroneously return the "Fig" label itself.

To be robust to index offsets like this, the algorithm now tries to generalize this XPath suffix using a similar approach to section 4.3.2. However, a key difference is that since we are generalizing the demonstration *target element*'s XPath suffix, we do not have a ground truth target element for each of the other parameter values. Therefore, we simply try to generalize the XPath suffix such that some target node is matched for each parameter value, and we opt to use classes and attributes which are semantically more meaningful than indices. For the example in Figure 4.4, the algorithm generalizes the target XPath suffix to be `> span.price`.

4.3.4 Limitations

4.3.4.1 Natural language understanding

The current algorithm does not leverage any natural language understanding (NLU) beyond simple text string matching. This means that if the user provides a parameter value that does not appear on the page, then no inferences will be made for that value.

4.3.4.2 Identifying rows and columns

The current approach for identifying table rows and columns looks for levels of the DOM where the children nodes have high similarity (note: this is to identify "semantic" tables, e.g., implemented with `divs`). If more than two levels of the DOM have high similarity scores, then our algorithm might choose the wrong two levels to use as its rows and columns. For example, the Forbes billionaires website [8] shows one semantic table (the hundreds of rows of billionaires), but the table is actually broken up by ads into 15-row subtables. Our algorithm currently identifies the 15-row subtables and the individual rows as the two levels with the highest similarity scores, therefore not considering the table columns in its inference.

[8] https://web.archive.org/web/20220401164932/https://www.forbes.com/billionaires/

4.3.4.3 Identifying parameter attributes from non-tabular hierarchically structured data

Our algorithm is currently not well-equipped to extract a *parameter-based* attribute from a list of entries, for example to answer questions like "What is the <*attribute*> of <*movie*>?" on the IMDb website[9], where <*attribute*> could be "rating", "duration", "gross", etc. This is because the algorithm currently assumes a set of attribute values will appear side-by-side as siblings or equivalent relatives. This is less often the case for non-tabular hierarchically structured data, for example, on the IMDb website, a movie's duration and genre are sibling nodes, but user rating, director, and votes appear in other parent nodes within a given entry.

4.3.4.4 Operating across multiple pages

The algorithm currently only operates on a single page of a website. It would be useful to support operations across multiple pages of a website, in particular searching for and performing superlative operations across results that are paginated (e.g., multiple pages of MLB players or movie titles).

4.4 User Study Setup

We conducted a lab study as a first step to assess the usability and usefulness of ParamMacros's natural language parameterization and program creation workflows.

4.4.1 Participants

We recruited 12 participants from our University mailing lists and Slack workspaces. Participants (5 female, 6 male, 1 non-binary) were ages 21–42 (median 28). At the time of the study 9 participants were students (1 undergraduate, 5 master's, 5 PhD), 1 a technology consultant, 1 a fundraising professional, and 1 a senior product manager. One participant reported no programming experience, three reported less than 1 year, two reported 1–2 years, two reported 2–5 years, one reported 5–10 years, and three reported more than 10 years of experience. The study lasted one hour and we compensated participants with a $25 USD Amazon gift card.

[9] https://web.archive.org/web/20220327010150/https://www.imdb.com/search/title/?count=100&groups=oscar_best_picture_winners&sort=year%2Cdesc&ref_=nv_ch_osc

4.4.2 Study Design

Our user study involved two meaningfully different sites: the Forbes billionares list[8] and an IMDb movie list[9]. The Forbes website included a table of the top 25 billionaires and their metadata (e.g., age, country, net worth), and enabled us to evaluate queries with multiple parameters. The IMDb website included a list of 25 movies and their metadata (e.g., rating, director, gross revenue), and enabled us to evaluate queries on non-tabular hierarchically structured data. We used replicas of the original sites in order to work around some of our system's inference limitations. The goal of this study was to understand how users interact with ParamMacros within the scope of inferences it supports. We used a between-subjects design. Participants were assigned to one of the two websites (six participants per website). The study included three stages:

4.4.2.1 Enumerating Queries

We showed each participant their assigned website and asked them to write 5 queries that could be objectively answered using the content on that website.

4.4.2.2 Parameterizing Queries

We showed participants a tutorial video parameterizing the query "For the person with the most home runs, how many did they have?" on the Major League Baseball website. We showed how to generalize "home runs" and "most" to parameters <*statistic*> and <*superlative*>, respectively. We then gave each participant three queries to parameterize: two queries they wrote themselves and one pre-determined query (identical across participants per given website)[10]. This allowed us to see variety in how people parameterize different queries, as well as observe patterns for a common query.

4.4.2.3 Creating a program

We showed participants a tutorial video creating a demonstration and validating the generated program. We then presented participants with two pre-made parameterized queries to create programs for. We chose to use pre-made queries to ensure they were domain-appropriate for the webpage, sufficiently challenging, comparable across users, and supported by our inference engine. The queries for Forbes were "What is the <*metadata*> of the <*most/least*> <*age/net worth*> billionaire in <*country*>?" and "What is <*person*>'s net worth?". The queries for IMDb were "What was the rating for <*movie*>?" and "What was the gross for the <*most/least*> grossing movie?".

[10] One participant per website did not complete the common pre-determined task due to an adjustment to the study design.

After participants completed all three stages, we administered a seven-point Likert scale survey regarding ease of use and usefulness, and conducted a semi-structured interview.

4.5 User Study Results

Overall, participants found ParamMacros's program creation process to be intuitive and useful. We found that the parameterization process is promising but some participants needed time before becoming comfortable with it.

4.5.1 Parameterizing questions

4.5.1.1 Parameterization patterns

The target webpage provided important context that helped ground participants' parameterizations. Participants often parameterized proper nouns, attributes, and numbers in questions. As an example, for the common question we presented for Forbes, "Who is the youngest billionaire in the United States?", all five[10] participants parameterized "youngest" to be a superlative and "United States" to be a country. For the common question for IMDb, "What was the rating for Nomadland?", all five[10] participants parameterized "Nomadland" to be a movie, and three of five participants parameterized "rating" to be an attribute, allowing alternative values such as "gross", "genre", and "runtime". Two participants also parameterized generic terms to allow more specific values, e.g., P9 parameterized "movie" to have alternative values "thriller" and "action".

In addition to using parameters to allow alternative values with different meanings, three participants created parameters to allow flexibility in word choice and phrasing. For example, P9 parameterized "How long" to also allow the value "What's the length of". These participants understood that "there is no one way to make a statement or to ask a question" (P7) and the potential implications of that.

Two participants commented that there were multiple granularities at which they could parameterize questions, and they were unsure what granularity to choose. For example, a coarse-grained parameterization of "What was the rating for Nomadland?" would simply parameterize "Nomadland" to any kind of "movie". A finer-grained parameterization would also parameterize "rating" to "attribute" (e.g., genre, gross), or even parameterize "What" to different question types.

4.5.1.2 Alternative values

Participants found auto-extracted alternative values useful when they matched the user's expectation. Participants commonly leveraged auto-extracted values when parameterizing proper nouns, e.g., movie titles (all six IMDb participants) and countries (five of six Forbes participants). This is likely because these proper nouns are distinct, so our algorithm was successful at finding them on the page.

In other cases, participants noticed that the extracted values were not meaningful or that no extracted values were returned. In these cases, participants just wrote their desired alternative values manually. To improve confidence amongst users and provide meaningful alternative values in more situations, future work should leverage natural language understanding to better interpret the website and parameter of interest, and should embed context alongside the candidate values to reveal their source (e.g., their location on the page).

4.5.1.3 Understandability

Participants had varying opinions on the parameterization workflow. Nine of 12 participants responded that they "somewhat agree" (5), "agree" (3), or "strongly agree" (1) on a seven-point Likert scale that the parameterizatiton workflow was easy to use. Some participants said it took them "some time to figure out what a parameter actually means" (P10) but that they better understood after seeing parameters applied later in the program creation stage.

4.5.2 Creating a program

All Forbes participants successfully created correct programs for each of the two program creation tasks (with the exception of P5, whose browser stopped working during the study). All IMDb participants successfully created correct programs for the "What was the gross for the <*most/least*> grossing movie?" task. Note that during the study we discovered an inference limitation in automating the other IMDb task ("What was the rating for <*movie*>?")—participants' programs returned the correct rating for some movies, but for others exhibited an off-by-one error, returning the rating for the next movie in the list.

Participants had largely positive feedback on the program creation process, saying it was "intuitive" (P2, P3) and that "starting the recording, clicking different areas, extracting, that made a lot of sense to me" (P1). 11 of 12 participants responded that they "somewhat agree" (4), "agree" (4), or "strongly agree" (3) on a seven-point Likert scale that the demonstration workflow was easy to use.

4.5.3 Usefulness

Participants were positive about the usefulness of the overall system. All participants responded that they "somewhat agree" (3), "agree" (6), or "strongly agree" (3) on a seven-point Likert scale that the system was useful for creating macros. Seven participants thought that these macros would be useful for answering questions about data in spreadsheets. One participant (P1) said for her work in fundraising she frequently asks questions like “Who’s giving the most?” when creating strategies for reaching out to donors. Two participants (P5 and P12) commented that they ask questions like “Which participant had the highest <*x*>, and how old were they?” in their user research. Two participants said they use intelligent voice assistants for personal tasks (e.g., playing music on Spotify, searching for bus routes) and would appreciate the ability to customize and correct errors.

4.5.4 Threats to Validity

Since we conducted a lab study and provided participants with predetermined websites, participants might not have had as intrinsic a motivation or understanding of meaningful questions to be asked or answered on the website, as compared with websites they encounter in the wild. In future work, it would be useful to study automation systems like ParamMacros in the wild to further assess usability and understand usage patterns.

4.6 Discussion and Future Work

Parameterizing natural language and creating a demonstration seems to be a promising approach for enabling end-users to create custom question-answering automation. Most of our user study participants were able to create meaningful question parameterizations and working programs. Although it took some participants some time to understand what parts of their questions made sense to parameterize, we believe this is a reasonable learning curve and suspect that end-users who already know the kinds of questions they want to automate will know what parameterizations are helpful.

In practice, there is diversity in how people may phrase the same question, but parameterized questions follow a very specific phrasing. To support natural speaking patterns, an important area of future work would be to use natural language understanding to map end-user freeform questions to the filled-in parameterized questions they best match.

Our current inference algorithm focuses on structural patterns in the website DOM to identify candidate parameter values and to generalize the user’s demonstration. This works for content that follows a consistent DOM structure, but has limitations if there is variation. Incorporating natural language understanding [90] would enable us to uncover semantic patterns that cannot be found based on structure alone, which would help identify alternative parameter values and more intelligently infer likely target elements. Regardless, there will always be edge case data or patterns in the DOM that an inference algorithm will not correctly understand. To still allow users to create custom automation in these situations, PBD systems may want to enable users to write small chunks of code to extract the desired data [94].

ParamMacros assumes the user largely wants to generalize the same behavior across all parameter values. If the user instead wants drastically different behavior in a particular situation, the user can create a refinement demonstration which simply just creates a different program to run in that situation. Future work should explore more holistic approaches for enabling the end-user to encode conditional logic, perhaps leveraging or building on approaches in Pumice [91].

4.7 Conclusion

We propose leveraging end-users to parameterize natural language queries that they want to create automation macros for. End-users know the kinds of questions they want their automation macro to support, so we leverage their understanding of their goals to identify meaningful parameters and possible values. A meaningful set of parameters and their values provides programming-by-demonstration systems a scope of the set of tasks they should support and hints on how to generalize. We designed a PBD system, ParamMacros, that applies this approach and enables end-users to create custom automation macros for answering questions about website content. End-users identify parameters in their natural language question and then create a demonstration of how to answer that question on the website. Results from our user study suggest that users can identify meaningful parameters in natural language questions and would find a parameterization and PBD workflow useful for their automation needs.

CHAPTER 5

ScrapeViz: Authoring and Visualizing Web Scraping Macros through Hierarchical Visual Representations and In-Context Data

5.1 Introduction

User interface (UI) automation macros can save users time and effort by performing digital tasks programmatically. Some automation is readily available through virtual assistants or pre-programmed macros (e.g., in iOS Shortcuts [37]), but for long-tail needs users will need to create their own custom macros. Traditionally people create custom macros by writing code, but this is infeasible for non-programmers and can be challenging even for programmers – as I showed in chapter 3, programmers need to interpret the third party target website's Document Object Model (DOM) [16] and they experience challenges such as identifying appropriate element selection logic. Cypress [14] and a prototype web automation IDE we built (chapter 3) provide some assistance to developers through linking code to UI context, helping them understand the correctness of their element selection logic and how their automation code impacts the target website's state.

To make creating automation macros more accessible to a broader set of end-users, researchers have explored leveraging programming-by-demonstration (PBD) [62, 93] so that users can create macros without writing program code. The user provides examples or demonstrations of the desired program behavior for a few concrete scenarios, and then the system infers a generalized program. Prior work (including my own ParamMacros, described in chapter 4) has used PBD to enable users to create macros for personal task automation [82, 95, 87, 88, 90, 91, 104, 107] and web scraping [55, 64, 106, 57] of user interfaces. Although PBD has made it easier for people to create automation macros, challenges still remain in making PBD usable [84]. Several of the challenges Tessa Lau describes [84] are related to the user not fully understanding how their PBD-generated program works.

I present *ScrapeViz*, a new PBD tool for creating distributed hierarchical web scraping macros, with a focus on helping users understand scraping across different website pages. *Distributed hierarchical* data [55] are data that are *hierarchical* (consisting of parent-child relationships) and *distributed* across multiple website pages. ScrapeViz provides users a high-level visual representation illustrating the action sequence and groups of website pages visited, and low-level tools to help users understand scraped data in the context of their page sources. To author, users interact with a website page normally, navigating to desired data and selecting text to scrape. Once the user provides two examples of a specific type of data to scrape (e.g., two actor names in a list on an IMDb movie page), ScrapeViz generalizes to try to scrape the rest of that kind of data on the page. ScrapeViz also generalizes navigation actions (e.g., clicking to open two movie title pages on the IMDb website will generalize to open all movie pages) and generalizes down-stream actions across each of these pages (e.g., automatically scraping actor names for all movie pages). During authoring, ScrapeViz automatically renders a storyboard-like visual representation of website pages visited (each page shown in a small window), illustrating parent-child relationships and groups of sibling pages. Each website page also highlights what data is scraped with color-coded borders. Finally, scraped data is presented in an interactive table, where the user can click on a cell to be taken to its source within a website page. This visual representation and interactive table serve to help authors and consumers alike – for authors to keep track of their actions and how they generalize across pages in real-time, and for consumers to understand high-level actions and data sources.

The key novelty of ScrapeViz is in the tools it offers for understanding web scraping behavior across multiple website pages. Prior systems are also tailored toward creating nested-loop macros [55, 64, 106, 57] or parameter-based macros [82, 88, 90, 107], but they do not specifically provide the user an overview of macro behavior across website pages. To understand how a macro behaves, a user can manually run their macro on different inputs, but this can be cumbersome. Some PBD systems [64, 106, 57, 107] give macro creators a preview of how the macro will run on the next input, which helps macro creators understand current behavior as they are building it. During authoring, Rousillon [55] uses color-coding to highlight corresponding UI elements on a given page, but this only reflects corresponding elements on a single and not multiple pages, and is present only during authoring and not afterwards. A key limitation of these approaches is that the user is only seeing the macro run on a small set of inputs – they are not getting an overview of how the macro works broadly and as a result may miss cases where the macro does not behave as desired. MIWA [57] provides an overview of macro behavior through a step-by-step natural language description of actions, visually highlighting corresponding UI elements on the website page for each action, and proactively pointing out potential anomalies. MIWA brings great advances, proving useful for participants in a user study, but MIWA is still lacking in three areas: 1) MIWA presents visual correspondence highlighting but for only one website page at a time, not enabling users to understand scraping behavior across multiple pages at a time; 2) MIWA's natural language description provides an overview of macro behavior, but lacks a glanceable visual overview of pages visited; and 3) users inspecting and debugging their output data table cannot determine precisely what website page and location each piece of data came from. Instead, ScrapeViz aims to help users get a broad overview of how their macro behaves across different page contexts through a visual representation of website pages visited and interactive output table that enables users to check the source of each datum. ScrapeViz's visual representation builds on ideas from our prototype web automation IDE (chapter 3) where we showed "before" and "after" UI states for each line of developer code, across loop iterations and user input values. ScrapeViz's visual representation provides a broader overview of macro behavior, allowing users to see all automation steps at the same time.

ScrapeViz leverages PBD approaches similar to prior PBD web scraping systems [55, 64, 106, 57]. With Rousillon [55], the macro author provides a single demonstration of how to scrape a single row of desired output data, and then the system infers a generalized macro. With WebRobot and its follow-on work [64, 106, 57], the macro author iteratively provides a sequence of demonstrations, and the system searches for a scraping pattern that matches all demonstrations to infer a generalized macro. ScrapeViz offers distributed hierarchical scraping capabilities similar to Rousillon's, while inferring from an iterative sequence of demonstrations (i.e., two) more similar to WebRobot.

We conducted a within-subjects lab study comparing ScrapeViz with Rousillon for reading and authoring tasks. Participants generally found both ScrapeViz and Rousillon easy to use and found benefits in aspects of each tool. Participants found ScrapeViz's interactive table especially helpful for understanding the source of scraped data and understanding anomalies. Participants also found ScrapeViz's illustration of actions performed and website pages visited helpful for understanding high-level behavior of macros they are consuming and for validating macros they are authoring in real-time.

The key contributions of this work are:

- A visual representation for distributed hierarchical web scraping macros, illustrating the sequence of actions and website pages visited, how these actions generalize across website pages, and scraped data highlighted within website pages.
- An interactive table approach for understanding scraped data in the context of its source website.
- ScrapeViz, a programming-by-demonstration tool for authoring distributed hierarchical web

scraping macros, which instantiates these visual representation and interactive table approaches.

- Findings from a lab study showing that participants found ScrapeViz easy to use for reading and authoring, and found its visual representation and interactive table helpful for understanding high-level actions and data sources, respectively.

Next I describe ScrapeViz's design, a sample usage scenario, its implementation and the scope of macros it supports, the results from a lab study, and a discussion and future work.

5.2 Design

ScrapeViz aims to help users more easily author and understand distributed hierarchical web scraping macros through programming-by-demonstration (PBD) and tools for visualizing macro behavior across website pages.

ScrapeViz has the following key design features:

- *Authoring*
  - *Programming-by-demonstration.* Authors interact as they would with a normal website page and simply need to provide two examples for each navigation or scraping action they want to perform. The system will then infer the rest of the UI elements to perform that action on, as described below.
  - *Live feedback.* As users author, they are shown immediate feedback (through the visualization tools described below) of actions performed, pages visited, generalizations across pages, and specific data scraped.
- *Visual representation.* To provide a concrete yet high-level overview of the macro's behavior.
  - *A storyboard-like visual* of automation action sequences and the hierarchy of resulting website pages, presented through multiple live viewports. Parallel website pages, navigated to from parallel elements on the parent website page, are grouped together to signify their semantic similarity and scraping generalizations across them.
  - *Color-coding* of UI elements identified as parallel based on user demonstrations.
  - *Interactivity* that enables users to zoom in on any given website page or group of pages to inspect or add additional actions. Also, when users scroll within a given page, all sibling pages will scroll in parallel to allow users to see corresponding scrape actions across pages.

- *Interactive output table* to enable users to easily discover the source of a particular scraped datum; users can click on a table cell and the interface will automatically take them to its data source, zooming in on the corresponding website page and highlighting the specific location the data was scraped from.

Below I present a sample usage scenario for ScrapeViz and how it could benefit users.

5.3 Sample Usage Scenario

5.3.1 Authoring

Susan is conducting a project analyzing the distribution of actor ages in popular movies and needs to collect data from movies on the IMDb website[1]. Specifically, for each movie listed, she needs to collect its name, its actors' names, and the birthdate for each actor[2]. Susan decides to use *ScrapeViz*[3] to create a web scraping macro to collect this data.

She first starts by **demonstrating collecting data for one movie and one of its actors**. The top-level page (Figure 5.1-A) includes a list of movie titles, each of which is a link leading to the individual movie's page. Susan clicks the title of the first movie on the page, Everything Everywhere All at Once, which scrapes the movie's title and navigates to the movie's page. Instead of opening the movie page in the current viewport, ScrapeViz leaves the current viewport intact at the top-level page (Figure 5.1-A), and creates a new smaller viewport (Figure 5.1-B) next to it to load the movie page in. This allows Susan to keep track of the pages she's visited and revisit them later to make edits or additions. Susan clicks on the smaller viewport to expand it and see the movie page more clearly; this then shrinks the original viewport containing the list of movie titles. The movie's page then includes a list of actor names, each of which is a link leading to the individual actor's page. She similarly clicks the first actor name on the Everything Everywhere All at Once movie page, Michelle Yeoh, which scrapes her name and then navigates to her page. Finally, she finds the birthdate "August 6, 1962" on Michelle Yeoh's page; she highlights the text and then clicks the "Scrape" button that pops up next to her cursor, and the interface adds a blue border around the birthdate indicating it has been selected (Figure 5.2-C). At each step of her demonstration, the interface builds up a storyboard-like visualization illustrating the sequence of pages she navigated to and the data scraping and clicking actions she performed, and also places the scraped text into the output table in the bottom-left corner (Fig 5.2). The visualization uses

[1] https://web.archive.org/web/20230404103018/https://www.imdb.com/search/title/?count=100&groups=oscar_best_picture_winners&sort=year%2Cdesc&ref_=nv_ch_osc

[2] IMDb example inspired by Rousillon [55]

[3] A demo video for ScrapeViz can be found at: https://dx.doi.org/10.7302/21952

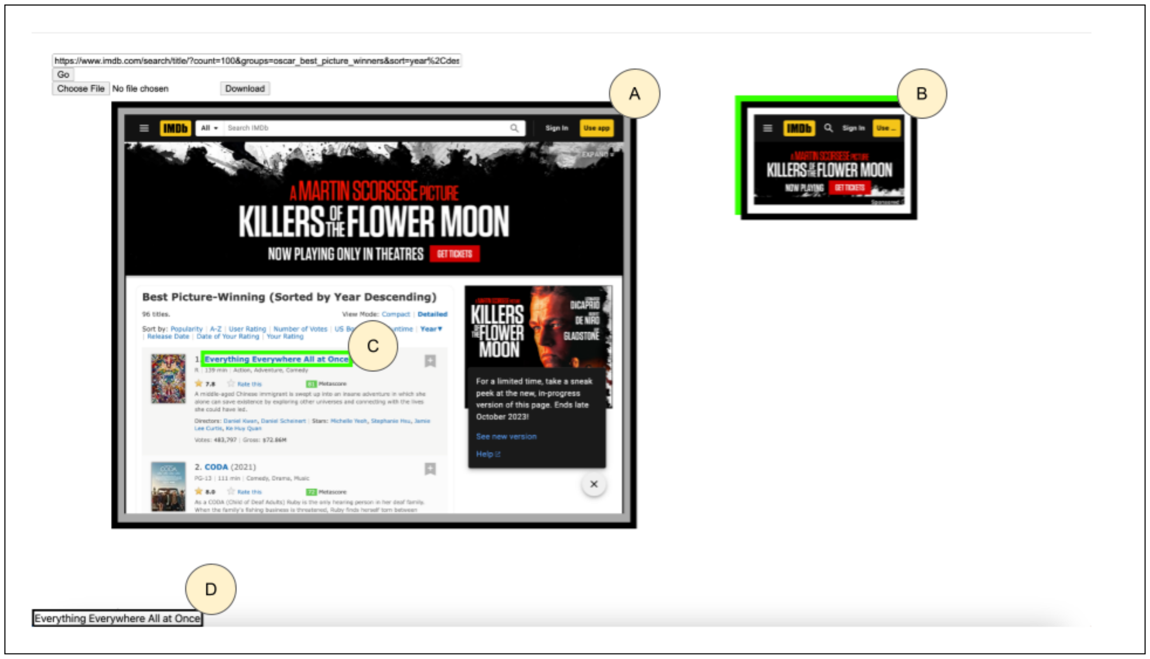


Figure 5.1: ScrapeViz after the user has performed one navigation/scraping action. Starting with just a single viewport presenting a list of Best Picture Winning movies on the IMDb website (A), the user clicks on the movie title “Everything Everywhere All at Once”. This opens a new, smaller viewport containing the movie page for “Everything Everywhere All at Once” (B), highlights the selected text on the original page with a green border (C), and adds the scraped text to the output table (D). Color-coding is used to indicate element interactions that lead to new UI states – for example here, the same color green border is used to convey that clicking the “Everything Everywhere All at Once” text (C) causes the browser to navigate to the next viewport (B).

color-coding to indicate element interactions that lead to new UIs – here, the same color green border is used to convey that clicking the “Everything Everywhere All at Once” text (Fig 5.2-A) causes the browser to navigate to page 2 (Fig 5.2-2), and similarly red is used for Michelle Yeoh’s name (Fig 5.2-B) leading to page 3 (Fig 5.2-3).

Next, Susan wants the same action sequence she performed for the actress “Michelle Yeoh” to also be performed for the other actors on the page – in other words, she wants to similarly scrape the name and birthdate for Stephanie Hsu, Jamie Lee Curtis, Ke Huy Quan, and so on. To tell the system to generalize in this way, **she simply needs to click on a second actor name**, e.g., “Stephanie Hsu” (Figure 5.3). The system then understands that Susan wants to perform the same kind of actions demonstrated for Michelle Yeoh for the other actors on the page too, which it illustrates through an updated visualization: a red border around each actor name on the Everything Everywhere All at Once page (Fig 5.4-A), a new small viewport for each actor page that is visited upon clicking each of the actor names added to the rightmost column (Fig 5.4-C), a blue border around the birthdate on each actor page (not pictured), and scraped actor names (Fig 5.4-B) and

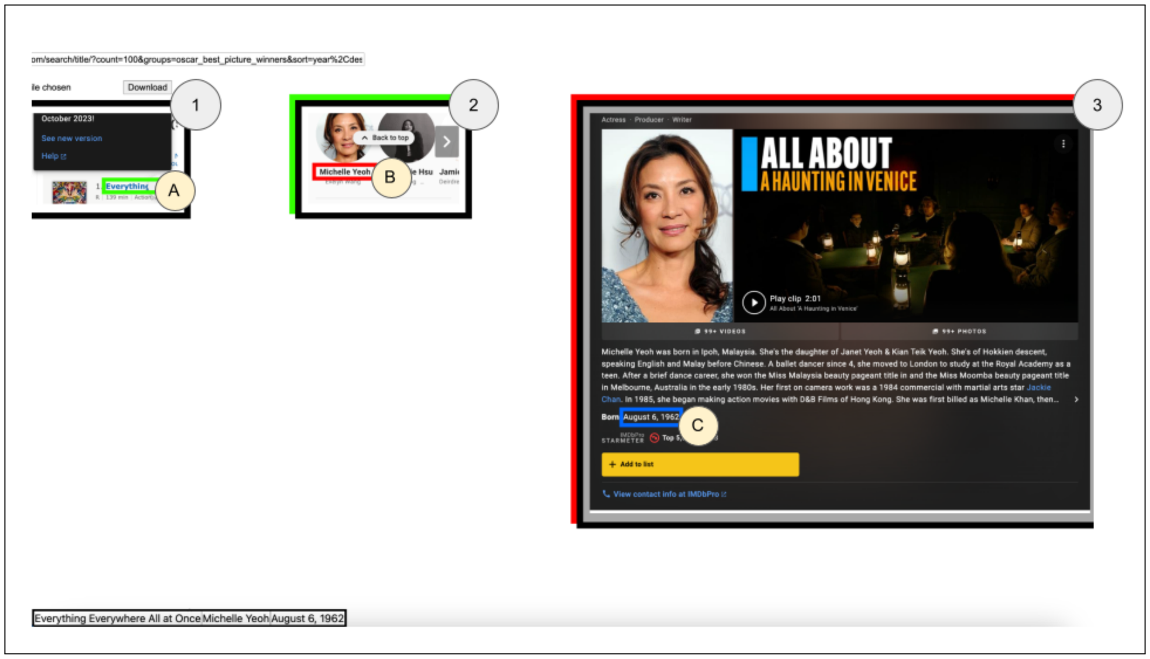
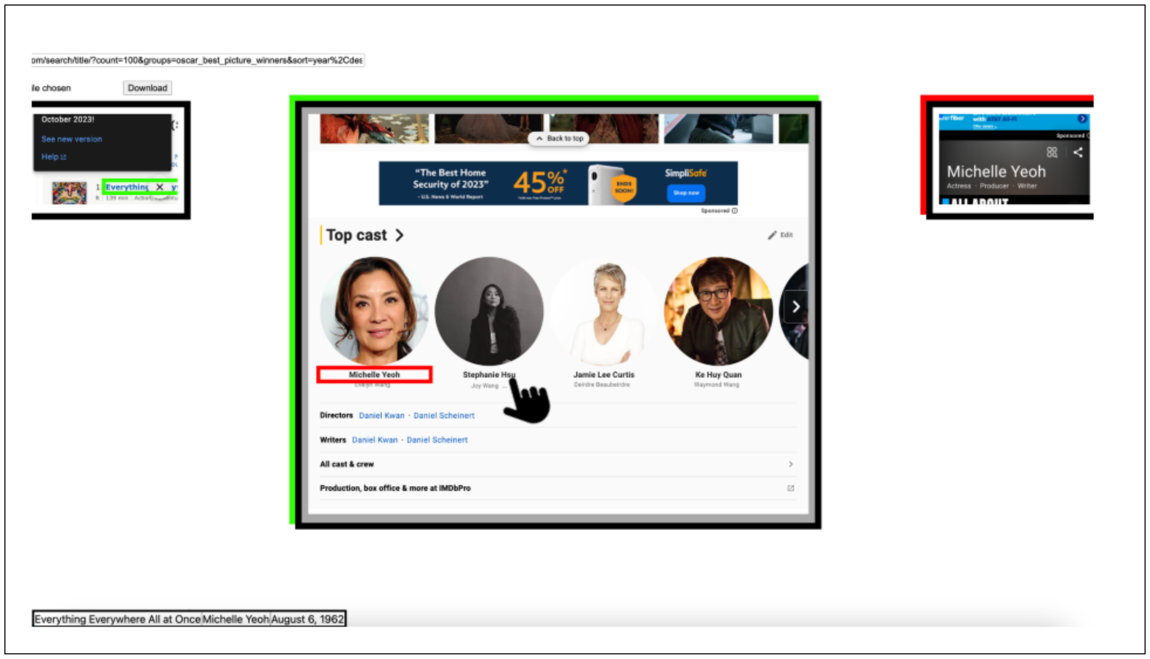

1
2
3
A
B
C
Everything Everywhere All at Once Michelle Yeoh August 6, 1962

Figure 5.2: A storyboard visualization generated as the macro author demonstrates a single scraping sequence, which illustrates the pages they visited and the scraping and clicking actions they performed: the author first clicked and scraped the movie title “Everything Everywhere All at Once” (A), causing the browser to navigate to the Everything Everywhere All at Once movie page (2); once on the Everything Everywhere All at Once movie page, the author clicked and scraped the actor name “Michelle Yeoh” (B), causing the browser to navigate to Michelle Yeoh’s page (3); once on Michelle Yeoh’s page, the author scraped birthdate text (C). Color-coding is used to indicate element interactions that lead to new UI states – for example here, the same color green border is used to convey that clicking the “Everything Everywhere All at Once” text (A) causes the browser to navigate to the next viewport (2).

actor birthdates (Fig 5.4-D) added to the output table.

Next, Susan similarly wants to specify that all of the action sequences performed for the Everything Everywhere All at Once movie should also be performed for each of the other movies on the page. She does this by clicking “CODA” to give a second movie example, which again results in an updated macro and visualization (Fig 5.5) – green borders around the movie titles on the top-level page (e.g., “Everything Everywhere All at Once”, “CODA”, “Nomadland”) indicate that they will be scraped and clicked like “Everything Everywhere All at Once” (Fig 5.5-A), and the resulting movie pages visited are shown in viewports in the middle column (Fig 5.5-B). The way that actor names were scraped and clicked for the Everything Everywhere All at Once movie page will automatically be generalized to the other movie pages, too, as evidenced by the red borders around actor names on all of the movie pages in the middle column (Fig 5.5-C).

In Figure 5.5-D note that some birthdates are not immediately shown in the output table and


Top cast
Michelle Yeoh
Stephanie Hsu
Jamie Lee Curtis
Ke Huy Quan
Directors Daniel Kwan · Daniel Scheinert
Writers Daniel Kwan · Daniel Scheinert
All cast & crew
Production, box office & more at IMDbPro
Download
Michelle Yeoh

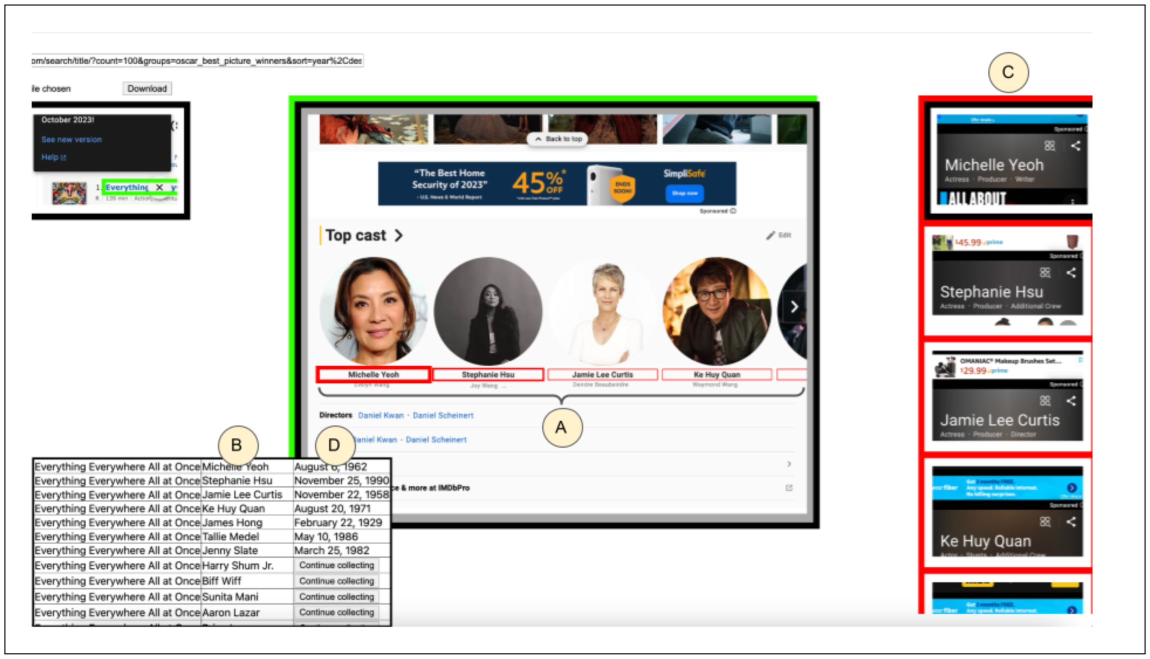
Everything Everywhere All at Once | Michelle Yeoh | August 6, 1962

Figure 5.3: Currently only one actor name is scraped – Michelle Yeoh. To scrape the rest of the actors' names, and additionally replicate scraping the actor's birthdate and perform any other downstream actions, the user only needs to give one other actor example, e.g., by clicking on "Stephanie Hsu".

instead "Continue collecting" buttons are shown. "Continue collecting" is presented in a cell when the expected website page where the generalized action (e.g., scrape birthdate) should be performed has not been rendered yet (i.e., the viewport has not been shown yet). There are a couple reasons why a particular website viewport may not be shown yet:

- More sibling webpages exist than the maximum (seven) that we will show at a time. For example, the first seven actor pages for Everything Everywhere All at Once are displayed (Figure 5.5-E), but no more after that. As a result, the page for Harry Shum Jr., the eighth actor, is not rendered, so his birthdate cannot be scraped and instead a "Continue collecting" button is shown in the birthdate column (Figure 5.5-D).

- The parent webpage that directs to this webpage has not yet been the **active** viewport (denoted with a thick black border), so its children webpages (including the one of interest) have not been rendered yet. For example, in Figure 5.6, birthdates for actors from "The Shape of Water" have not yet been scraped (Figure 5.6-C) because the movie viewport for "The Shape of Water" (Figure 5.6-B) has not yet been active, so actor pages for "The Shape of Water" have not yet been rendered in the child column (Figure 5.6-E). The actor pages shown in column E will only ever be for actors from the current active movie, "Everything

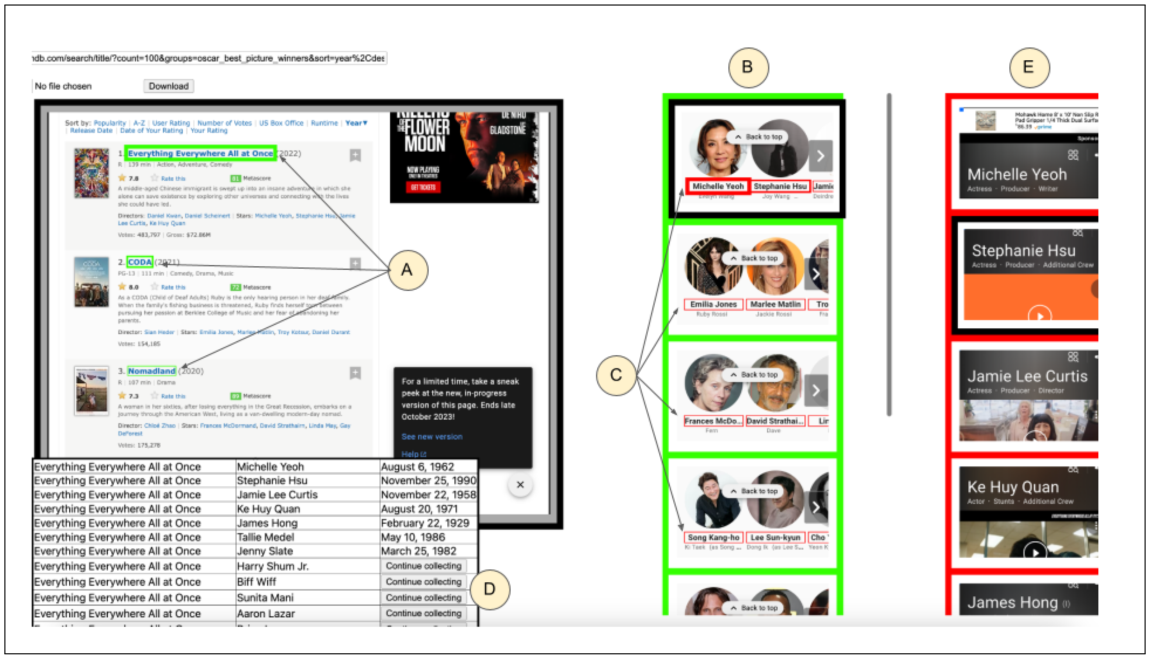


Everything Everywhere All at Once	Michelle Yeoh	August 6, 1962
Everything Everywhere All at Once	Stephanie Hsu	November 25, 1990
Everything Everywhere All at Once	Jamie Lee Curtis	November 22, 1958
Everything Everywhere All at Once	Ke Huy Quan	August 20, 1971
Everything Everywhere All at Once	James Hong	February 22, 1929
Everything Everywhere All at Once	Tallie Medel	May 10, 1986
Everything Everywhere All at Once	Jenny Slate	March 25, 1982
Everything Everywhere All at Once	Harry Shum Jr.	Continue collecting
Everything Everywhere All at Once	Biff Wiff	Continue collecting
Everything Everywhere All at Once	Sunita Mani	Continue collecting
Everything Everywhere All at Once	Aaron Lazar	Continue collecting

Figure 5.4: After the user provided a second actor example (in Figure 5.3), “Stephanie Hsu”, ScrapeViz automatically generalizes to scrape the rest of the actors’ names on the movie page (signified through a red border around each actor’s name (A); the scraped actor name also appears in the second column of the output table (B)), click on those names to open each actor’s page in a new small viewport in the right column (C), and scrape the birthdate from each actor page (stored in the third column of the table (D)).

Everywhere All at Once” (Figure 5.6-D).

Upon clicking a “Continue collecting” button, ScrapeViz will bring the viewport containing the expected data into view and make it the active viewport, and also bring all of its ancestor viewports into view (as briefly explained above, ScrapeViz only shows a given downstream viewport if its ancestors are currently selected). If the user were to click the “Continue collecting” button next to Harry Shum Jr.’s name (Figure 5.5-D), ScrapeViz will update to bring Harry Shum Jr.’s page into view (Figure 5.7-A), as well as the next five actors’ pages (Figure 5.7-B), and scrape their birthdates and place them in the output table (Figure 5.7-C). If the user were to click the “Continue collecting” button next to Sally Hawkin’s name (Figure 5.6-C), ScrapeViz will update to select the viewport for “The Shape of Water” in the second column, denoted by a black border (Figure 5.8-A). This will then result in rendering The Shape of Water’s actor pages in the right-most column (Figure 5.8-B), making Sally Hawkin’s page active (Figure 5.8-C), and scraping and placing these actors’ birthdates into the output table (Figure 5.8-D). To achieve the same effect, the user alternatively could have clicked the next button (e.g., as in figure 5.7-D) to bring the next set of “Everything Everywhere All at Once” actor pages into view, or clicked on a specific movie



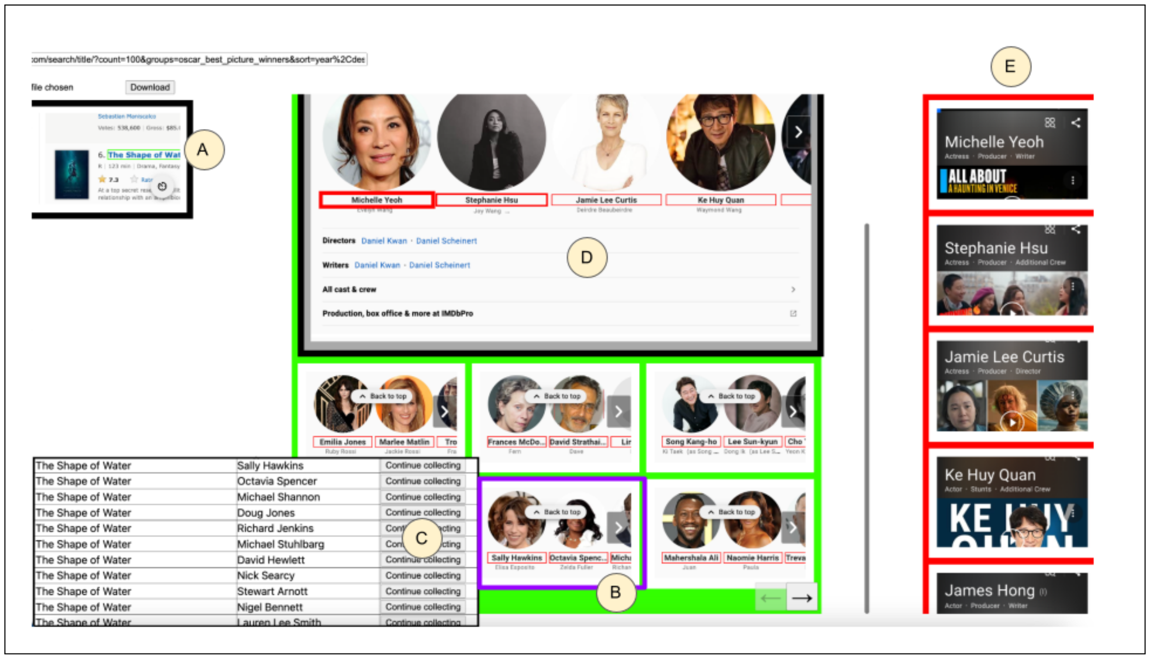
Everything Everywhere All at Once	Michelle Yeoh	August 6, 1962
Everything Everywhere All at Once	Stephanie Hsu	November 25, 1990
Everything Everywhere All at Once	Jamie Lee Curtis	November 22, 1958
Everything Everywhere All at Once	Ke Huy Quan	August 20, 1971
Everything Everywhere All at Once	James Hong	February 22, 1929
Everything Everywhere All at Once	Tallie Medel	May 10, 1986
Everything Everywhere All at Once	Jenny Slate	March 25, 1982
Everything Everywhere All at Once	Harry Shum Jr.	Continue collecting
Everything Everywhere All at Once	Biff Wiff	Continue collecting
Everything Everywhere All at Once	Sunita Mani	Continue collecting
Everything Everywhere All at Once	Aaron Lazar	Continue collecting

Figure 5.5: Full visualization of the generalized macro's behavior after the author has generalized for both actor names and movie titles.

viewport (Figure 5.6-B) to make that viewport active and bring its child actor pages into view.

After browsing through the website pages and inspecting the output table, Susan feels confident that the macro is scraping the data she wants. ScrapeViz has allowed Susan to author **nested-loop scraping logic** and across multiple pages – *for each movie*, scrape its name and click on it to reach its list of actors; *for each actor*, scrape their name and click on it to reach their birthdate and scrape their birthdate text.

Note that we presented only one way of the many possible ways Susan could have authored this particular IMDb scraping macro. In this example here, we first authored a single interaction sequence – scraping one movie title, then one actor name, then the actor's birthdate (Figure 5.2). Then we incrementally generalized actions in the reverse order – generalizing actors first and then movies second. However, **any order for initial or generalization interactions is allowed** – for example, the user could instead have started by scraping two movie titles, which immediately would have generalized the movie title scraping action to all movies on the page; the user could then have gone to one of the many movie pages shown and scraped two actor names, which would have generalized to scrape all actors on that page, as well as all actors across all movie pages.


A
B
C
D
E
Download
Michelle Yeoh
Stephanie Hsu
Jamie Lee Curtis
Ke Huy Quan
James Hong
Directors Daniel Kwan · Daniel Scheinert
Writers Daniel Kwan · Daniel Scheinert
All cast & crew
Production, box office & more at IMDbPro
Back to top

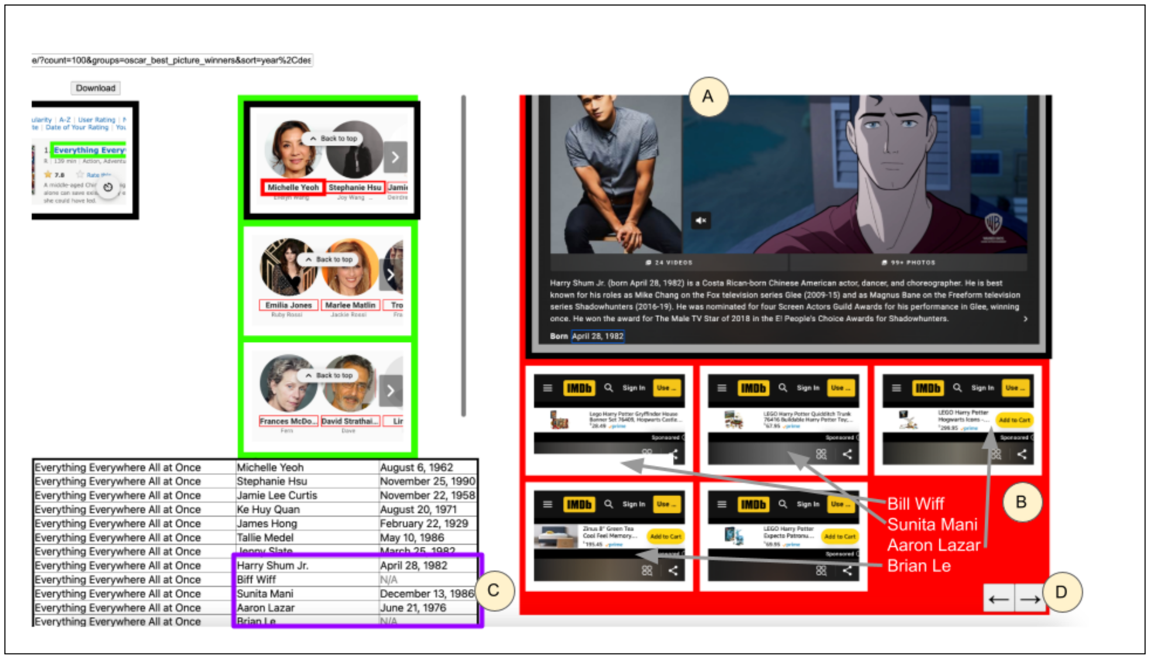
The Shape of Water	Sally Hawkins	Continue collecting
The Shape of Water	Octavia Spencer	Continue collecting
The Shape of Water	Michael Shannon	Continue collecting
The Shape of Water	Doug Jones	Continue collecting
The Shape of Water	Richard Jenkins	Continue collecting
The Shape of Water	Michael Stuhlbarg	Continue collecting
The Shape of Water	David Hewlett	Continue collecting
The Shape of Water	Nick Searcy	Continue collecting
The Shape of Water	Stewart Arnott	Continue collecting
The Shape of Water	Nigel Bennett	Continue collecting
The Shape of Water	Lauren Lee Smith	Continue collecting

Figure 5.6: Although the macro has now been generalized to scrape all movie titles, data for some movies and actors still has not been added to the table yet. For example, "The Shape of Water" has been scraped from the top-level page (A) and its movie page (B) has been rendered in the middle column showing actors' names (e.g., Sally Hawkins, Octavia Spencer), but birthdates for these actors are not actually shown in the output table yet and instead "Continue collecting" buttons are shown (C). This is because the actors' individual pages have not actually been rendered yet, so the macro has not had an opportunity to scrape from them yet. ScrapeViz shows child viewports only for the active viewport a given level. Here, the active viewport in the middle column (movie pages) is the "Everything Everywhere All at Once" page (D), so the child viewports shown in the rightmost column are only for actors from "Everything Everywhere All at Once" (E) and not any other movies. To render "The Shape of Water" actors in column E and scrape their birthdates, the user can either click "The Shape of Water" viewport (B) to make it active or click on a "Continue collecting" button in the table (C).

5.3.2 Consuming

Imagine Susan takes a break from this project. Several months later she returns to her macro wanting to use it again to collect data for movies recently added to the IMDb website. She wants a reminder of exactly what the macro does since she has forgotten. She can simply review the interactive visual representation to see the sequence of actions performed and website pages visited, and what exact data it collects in the table.

ScrapeViz would likely also prove useful for macro consumers who did *not* author the macro themselves. If Susan shares her macro with a colleague, her colleague can use the macro visualization to get a quick overview of what the macro does without needing to ask Susan. The visual

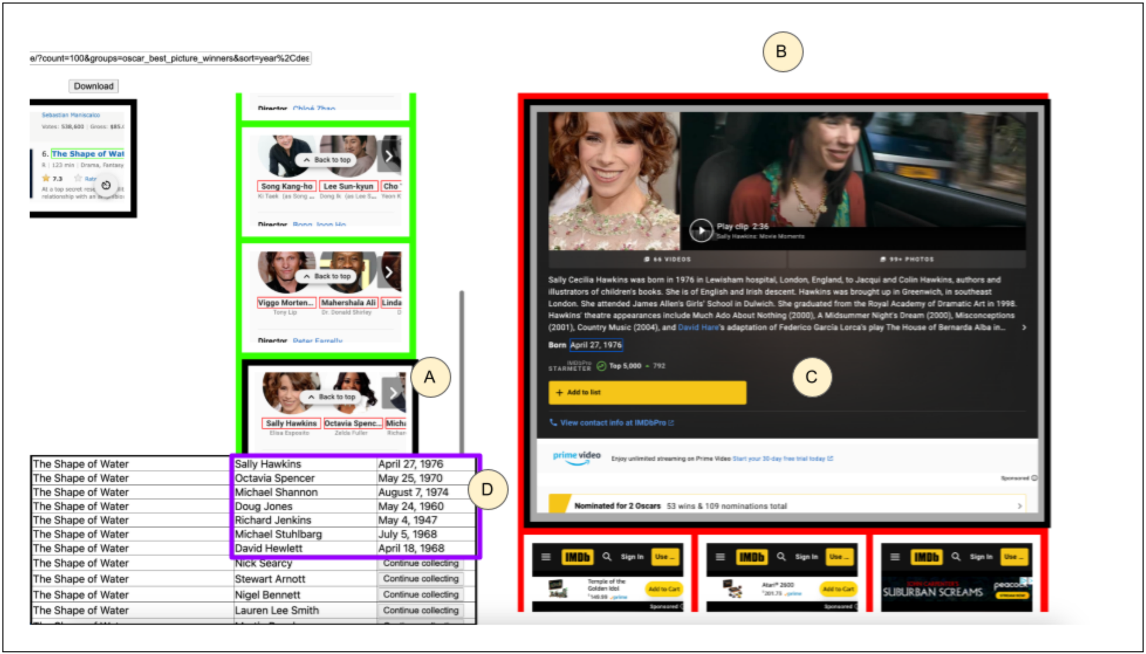


Everything Everywhere All at Once	Michelle Yeoh	August 6, 1962
Everything Everywhere All at Once	Stephanie Hsu	November 25, 1990
Everything Everywhere All at Once	Jamie Lee Curtis	November 22, 1958
Everything Everywhere All at Once	Ke Huy Quan	August 20, 1971
Everything Everywhere All at Once	James Hong	February 22, 1929
Everything Everywhere All at Once	Tallie Medel	May 10, 1986
Everything Everywhere All at Once	Jenny Slate	March 25, 1982
Everything Everywhere All at Once	Harry Shum Jr.	April 28, 1982
Everything Everywhere All at Once	Biff Wiff	N/A
Everything Everywhere All at Once	Sunita Mani	December 13, 1986
Everything Everywhere All at Once	Aaron Lazar	June 21, 1976
Everything Everywhere All at Once	Brian Le	N/A

Figure 5.7: After clicking the "Continue collecting" button next to Harry Shum Jr.'s name in Figure 5.5, ScrapeViz updates to bring Harry Shum Jr.'s page into view (A), as well as the next five actors for Everything Everywhere All at Once (B), and scrapes and places their birthdates into the output table (C).

representation would allow the colleague to get an overview of the macro's behavior without needing to watch a long and linear execution of the macro, where it may be harder to keep track of which pages were visited and which data was scraped. Beyond simply looking at the high-level sequence and groups of website pages, Susan's colleague can also look more closely at the exact data collected. First, if she wants to understand exactly what certain data in the table mean and where they came from, she can click on a cell to be taken directly to the source page. For example, the colleague can click on the cell containing "February 22, 1929" (next to the cell containing "James Hong" in Figure 5.7), and ScrapeViz will automatically bring actor James Hong's page into view in the rightmost column, and specifically scroll to and highlight the text "February 22, 1929" (which follows the text "Born"), making it clear to the colleague that this is the actor's birthdate. Clicking on cells in the table may be especially helpful for users who notice an anomaly in the table data. For example, imagine the colleague notices the "N/A" in the birthdate cell for Bill Wiff (Figure 5.7-C) and she wants to understand why there is an N/A instead of a date. She can click on this "N/A" and ScrapeViz will bring Bill Wiff's page into view, where she will notice that no birthdate is listed.

As the colleague works to understand the data scraped, she likely will want to explore the different website pages. We have already mentioned several interactions for navigating: clicking


e/?count=100&groups=oscar_best_picture_winners&sort=year%2Cdes
Download
B
A
C
D
Sally Hawkins
Octavia Spenc...
Back to top
Play clip 2:36
Sally Hawkins: Movie Moments
Add to list
View contact info at IMDbPro
Nominated for 2 Oscars

Sally Cecilia Hawkins was born in 1976 in Lewisham hospital, London, England, to Jacqui and Colin Hawkins, authors and illustrators of children's books. She is of English and Irish descent. Hawkins was brought up in Greenwich, in southeast London. She attended James Allen's Girls' School in Dulwich. She graduated from the Royal Academy of Dramatic Art in 1998. Hawkins' theatre appearances include Much Ado About Nothing (2000), A Midsummer Night's Dream (2000), Misconceptions (2001), Country Music (2004), and David Hare's adaptation of Federico Garcia Lorca's play The House of Bernarda Alba in...

Born April 27, 1976

The Shape of Water	Sally Hawkins	April 27, 1976
The Shape of Water	Octavia Spencer	May 25, 1970
The Shape of Water	Michael Shannon	August 7, 1974
The Shape of Water	Doug Jones	May 24, 1960
The Shape of Water	Richard Jenkins	May 4, 1947
The Shape of Water	Michael Stuhlbarg	July 5, 1968
The Shape of Water	David Hewlett	April 18, 1968
The Shape of Water	Nick Searcy	Continue collecting
The Shape of Water	Stewart Arnott	Continue collecting
The Shape of Water	Nigel Bennett	Continue collecting
The Shape of Water	Lauren Lee Smith	Continue collecting

Figure 5.8: After clicking the "Continue collecting" button next to Sally Hawkin's name in Figure 5.6, ScrapeViz updates to selecting the Shape of Water's viewport, denoted by the black border (A). This then results in rendering The Shape of Water's actor pages in the rightmost column (B), making Sally Hawkin's page active (C), and scraping and placing these actors' birthdates into the output table (D).

directly on a viewport to expand it, clicking on a cell in the table to be taken to the source page and location, and clicking on the "next" and "previous" buttons of a group to see other pages in that group. One other interaction for navigating is clicking on a selected element in a website page to be taken to the resulting child page – just as you would normally click an element link in a regular browser window. For example, the colleague can click on the selected element "Nomadland" (Figure 5.5) in the top-level page and ScrapeViz will make the resulting child, the Nomadland movie page, active in the second column of the visual representation.

5.3.3 Editing

Now that the colleague has worked to understand the macro, she understands whether it meets her needs. Imagine she wants to conduct an analysis of actors' birthplaces. On IMDb's actor pages, the actor's birthplace is typically included in their biographical text, so the colleague wants to adapt the macro to scrape each actor's biographical text. She does not need to start from scratch, and instead can simply add this new scrape action to the existing macro. To do this, she simply needs to provide an example of scraping the biographical text for *one* actor page. Once she provides an example on one actor page, this biographical text scrape action will automatically be replicated across all actor pages. This is the key idea underlying generalization in ScrapeViz – the actions performed on one page will be replicated across all sibling pages.

The colleague can also remove actions from the macro as appropriate. For example, since her goal now is simply to scrape an actor's biographical text, she no longer needs to scrape the birthdate. If she wishes, she can delete the birthdate scrape action by hovering over the birthdate on any actor page and clicking the "x" button that appears. This will automatically delete the birthdate scrape action across all actor pages.

5.4 Scope of macros supported

ScrapeViz supports nested loop UI automation macros capable of scraping and clicking across parallel UI elements and parallel pages. ScrapeViz is a research prototype and therefore there are kinds of automation logic it does not support, though would be important for real-world use:

- Looping where the UI elements operated on appear on a linear sequence of pages – e.g., iteratively clicking the "Next" button to navigate through multiple pages of search results. The current ScrapeViz interaction would not support this, as two elements need to appear on the same page for the user to provide those as two parallel examples to generalize over.

- Parameter-based macros. To support these we would likely need to add a new interaction for users to specify which actions are informed by parameters. Alternatively, parameters may correspond to values typed into textfields – specifying this would require a different interaction than providing examples to generalize over. A different visualization model may also be needed since possible input values for the textfield are not known.

- Conditional logic. This would require different specification and visualization models and it would be important to ensure they are understandable to novice programmers.

5.5 Implementation

5.5.1 Inference

ScrapeViz leverages website structure-based inference methods from my prior work, ParamMacros (chapter 4). ScrapeViz is based on two kinds of generalization:

- **Generalizing across two example UI elements.** When the user clicks or scrapes a new UI element, we check if it may generalize with any UI element the user has previously clicked or scraped. I use my prior approach from ParamMacros where we leverage structural patterns in the website DOM [16] to search for a generalized *XPath formula* that matches all specified UI elements. Specifically, we look for a generalized XPath formula that matches the two elements of interest (e.g., the newly scraped "Stephanie Hsu" and the previously scraped "Michelle Yeoh" from Figure 5.3). This formula must have the form `/prefix/index/suffix`, namely, the only difference between the two elements' XPaths being the index of a single node. If a generalized XPath of this form is found, we then use that formula to enumerate the other matching elements on the page. For example, the index-based XPath for "Michelle Yeoh" is `/html/.../div[`**`1`**`]/div[2]/a` and for "Stephanie Hsu" is `/html/.../div[`**`2`**`]/div[2]/a`, so a generalized XPath that matches both of them is `/html/.../div[`**`index`**`]/div[2]/a`, which indeed does follow the required `/prefix/index/suffix` form. This generalized XPath then matches each of the other actor names on the page, with each numeric index of `index` matching a different actor.

- **Generalizing across pages.** We also generalize actions across pages. For example, in Fig 5.4, the user had scraped two example actor names from the Everything Everywhere All at Once movie page (Michelle Yeoh and Stephanie Hsu) and ScrapeViz generalized to select all actor names on that page. Later, after the user has generalized the macro to scrape all movie titles from the top-level page (Fig 5.5-A) and open viewports for each movie page (Fig 5.5-B), ScrapeViz now generalizes to replicate all actions from the original Everything Everywhere All at Once page to each of the other movie pages in column B. Specifically, ScrapeViz generalizes to select all actors from each of these pages, simply by applying the same generalized XPath formula `/html/.../div[index]/div[2]/a` to each of these other movie pages.

5.5.2 Interface

ScrapeViz is implemented as an Electron desktop app [18]. This supports a key requirement – the ability to embed multiple live website viewports at a time. We specifically use Electron's WebView component [40], which is a wrapper around Chromium's and enables rendering any target website url.

We render a React [33] web UI within the Electron window. The React UI contains WebViews, colored borders for each WebView and group, and an interactive table. We instrument WebViews with our own JavaScript logic to listen for user click and scrape demonstrations, identify new generalizations, and apply click and scrape generalizations to the page. When the user clicks on a link within a WebView (e.g., clicking the "Everything Everywhere All at Once" link in the top-level page of Figure 5.1), we intercept the resulting `will-navigate` [41] Electron event to prevent the existing WebView from navigating to the new url. Instead, we render the new url (the movie page for "Everything Everywhere All at Once") in a new WebView.

We keep a React data structure that keeps track of parent-child url relationships, and the specific single or generalized actions demonstrated for at a given url. Note that a given url can have any arbitrary number of actions performed on it (e.g., scraping not just the actor's birthdate in the IMDb example, but also the actor's biographical text). When the user provides a new demonstration or generalization within a WebView, the WebView sends a message back to the React app layer, which then updates its data structure with the new action. Each time the React app re-renders its UI, it also makes sure to replicate actions across parallel urls. This includes not only siblings but also "relatives"; e.g., for the IMDb example, scraping the birthdate is first demonstrated on Michelle Yeoh's page and replicated across the other Everything Everywhere All at Once actor pages, but it's also replicated across **all** the other movies' actor pages.

When a WebView is passed actions to perform, it searches for elements on the page matching the given concrete XPath (for a single-demonstration action) or the XPath formula (for a generalized action). It then adds a border overlay to each matched element and sends a message back to the React app with the scraped data. If the given action is a click action, it will also programmatically click each matched element, which will trigger opening a new WebView for any new urls.

5.6 Study Design

We conducted a within-subjects lab study to evaluate the usability and usefulness of ScrapeViz. We believe the key novelty of ScrapeViz is in its visual representation of macro actions and pages visited, how actions generalize across pages, and seeing scraped data in context of their source. While ScrapeViz leverages PBD, a number of other PBD tools already exist for web scraping (e.g., [55, 64, 106, 57]), and in particular Rousillon [55] which similarly enables distributed hierarchical web scraping. Therefore, we felt one of the more meaningful evaluations we could conduct would be in comparing ScrapeViz to another PBD web scraping tool, namely Rousillon, so that we could specifically evaluate the potential benefits and challenges of our visualization approach.

5.6.1 Participants

We recruited 12 participants (six men, six women; aged 22–52, median 28.5 years) from our university Slack workspaces and email lists. Participants came from a number of backgrounds: two data analytics engineers, one data engineer, one software engineer, two product/UX designers, one librarian, one IT manager, two computer science masters students, one information masters student, and one public policy research assistant. Participants had varying levels of programming experience: one participant with no programming experience, three participants with less than 1 year, two participants with 1–2 years, four participants with 2–5 years, and two participants with 5–10 years. Six participants had at least some experience with web scraping. We compensated participants with a $40 Amazon gift card for their time.

5.6.2 Protocol

Our study consisted of two types of tasks: reading and authoring. These were meant to emulate users *authoring* their own custom web scraping macros and users *reading* existing web scraping macros to reuse themselves, e.g., that a colleague shared with them or that they found online. We had participants complete one reading task per each of the two tools (ScrapeViz and Rousillon) and one authoring task per tool. Note that task website, tool used, and task order were counterbalanced across participants. Also note that we did not inform participants until the very end of the study which tool we built and which one was the control condition; we simply told participants both tools are research prototypes, so as a result we believe we received unbiased responses with regard to comparisons between the two tools.

5.6.2.1 Rousillon

We chose to compare ScrapeViz to Rousillon [55] because Rousillon is the closest existing PBD tool to ScrapeViz, supporting the same kind of task – distributed hierarchical web scraping. Rousillon uses a similar demonstration-based approach to ScrapeViz, but only requires one example item rather than two to generalize from. For example, in the context of the IMDb example (Figure 5.5), the user would begin recording and then only need to scrape one movie title, one actor for that movie, and the actor's birthdate. To indicate that they want to select text, the user needs to press their Alt or Option keyboard key and then click with their mouse on the desired UI element.

After the user stops recording, Rousillon will start to generate a generalized macro from the sequence of single examples. It will present to the user a block-based nested-loop program describing the generalized macro. The program's variable names are generic by default (e.g., `list_item_1`) but the user can manually edit the names to be more meaningful. Note that for our reading task we wrote more meaningful semantic names ahead of time. We felt this provided a more fair comparison for Rousillon and reflects how people typically consume programs written by others. Rousillon also presents a set of "relevant tables" to the user, to see the data extracted from each website page at demonstration time and their relationships; these tables often include more information than the user actually scraped themselves, but is used for interpreting new website pages and relevant data. For example, for the IMDb website, the relevant tables will include: 1) a table for the top-level page, for each movie title listed a row containing the movie title, its genre, a synopsis, its rating, etc, and 2) a table for a given movie's page, for each actor listed a row containing the actor name. When the user clicks the "Run" button to actually run the macro, Rousillon will open a new browser window and begin running the program linearly, and showing the user only one viewport at a time – opening a new tab for each new page visited and then closing the tab when it has finished visiting that page and its children.

When we presented participants with reading tasks using Rousillon, we gave them the Rousillon interface and a separate browser window containing the target website, in case participants wanted to freely explore the website alongside the given Rousillon block-based code and metadata before actually running the macro. We intentionally did not run the macro ahead of time because we wanted to give users the opportunity to observe the macro running linearly and see the different website pages visited. As the macro runs it adds new rows to an output data table. In addition to the data scraped by the user, this output table includes columns on the right indicating the loop indices for the given row, e.g., for the first movie "Everything Everywhere All at Once" and its first actor "Michelle Yeoh", the indices would be `1` and `1`; for the first movie "Everything Everywhere All at Once" and its second actor "Stephanie Hsu", the indices would be `1` and `2`; for the `i`th movie and its `j`th actor, the indices would be `i` and `j`.

5.6.2.2 Reading

We had participants complete one reading task using ScrapeViz and one using Rousillon. We used two websites (WTA[4] and Wayfair[5], described more below) and counterbalanced website and tool pairing and task orders across participants.

Tutorial: Before each reading task, we asked participants to watch a seven minute tutorial video for the given tool and answered any questions they had afterward. Note that in these videos we gave a tutorial of how to *author* using each tool, rather than simply how to read. The intent of this was to help participants understand the concept of *generalization* and give context for how the scraping results they would be shown were produced. The tutorial demonstrated a sample scraping task for scraping data from the IMDb website (like described in section 5.3): a list of the displayed movies, navigating into each movie's page to get a list of its actors, and navigating into each actor page to get the actor's birthdate. In addition to teaching the core demonstration and generalization features of each tool, we also taught the following:

- ScrapeViz: clicking on cells in the interactive table to be taken to its page source; clicking on a scraped element link within a website page to navigate to its resulting child page

[4]https://web.archive.org/web/20231012210012/https://www.wtatennis.com/stats

[5]https://web.archive.org/web/20220324013917/https://www.wayfair.com/furniture/sb0/sectionals-c413893.html

- Rousillon: block-based code; relevant tables which illustrate the data extracted at demonstration time that helped determine the generalized program

Task websites: We chose two websites for our reading tasks: WTA women's tennis and Wayfair furniture. We aimed to choose websites and associated scraping macros that were distinct in their semantic actions yet comparable in their difficulty. We intentionally chose macros that worked for the most part but also exhibited edge cases (e.g., missing or incorrect data for some entries), with the goal of requiring users to carefully inspect the macros and websites to identify these anomalies.

- WTA women's tennis: We presented participants with a macro that scrapes players' last names from a top-level statistics page that has a list of players in a tabular format. The macro also visits each individual player's page and scrapes their age and their coach's last name. Some players do not have a coach listed. When the coach's name is not listed, ScrapeViz presents an N/A in the output table while Rousillon presents an incorrect value (a "number of losses" number taken from elsewhere on the page).

- Wayfair furniture: We presented participants with a macro that scrapes furniture products' names from a top-level catalog page with a list of furniture products. The macro also visits each individual furniture product's page and scrapes a list of "variations" (e.g., fabrics, colors, orientations). However, it specifically will scrape these variations only if they are visually presented as a thumbnail (sometimes variations are instead hidden in a pop-out panel, and our macro does not scrape variations in this case). When variations are not shown as a thumbnail, ScrapeViz presents a single N/A in place of variations while Rousillon does not include that product in the output table at all.

Task instructions: We then gave participants an instructions document (which we also read aloud to them) with the following questions to answer:

1. What specific data are getting scraped? Feel free to explore with the tool and/or browser to confirm your understanding.

2. Is the data in the table correct, or do you see any errors or anomalies?

3. If you see any errors or anomalies, could you try to reason about why these are occurring? Note that these extraction algorithms can be brittle, so may work differently across different pages.

Survey and interview: After each task we administered a 7-point Likert scale survey. Finally, after completing both tasks, we conducted an interview to learn about participants' experience using each tool for reading – in particular asking how they worked to understand what data was being scraped and what anomalies existed, as well as what they thought of the unique features of each tool.

5.6.2.3 Authoring

We had participants complete one authoring task using ScrapeViz and one using Rousillon. We used two websites (Google Scholar[6] and Yelp[7], described more below) and counterbalanced website and tool pairing and task orders across participants.

Refresher and tutorial task: Before each authoring task, we gave participants a brief refresher of the key features of the given tool and answered any questions they had. Then, we had participants complete a brief tutorial task to ensure participants understood how to scrape using the tool. We presented them with a people directory website and asked them to simply scrape the names of the people (which all appear on the same page). This was to ensure participants were familiar with both the specific scraping interaction (Alt/Option key + click for Rousillon, and selecting text + clicking the "Scrape" button for ScrapeViz) as well as the demonstration and generalization paradigm (one example for Rousillon and two examples for ScrapeViz). We answered any questions participants had and helped them reach a correct solution if they could not on their first attempt.

Task websites and instructions: We chose two websites for our reading tasks: Google Scholar and Yelp. We aimed to choose websites and associated scraping macro tasks that were distinct in their semantic actions yet comparable in their difficulty. We intentionally chose relatively straightforward yet non-trivial tasks for which both ScrapeViz and Rousillon would support generalization. Since participants were new to the tools, we wanted them to focus on the key concepts of demonstration and generalization without needing to worry about edge cases.

We asked participants to scrape the following data:

- Google Scholar: researcher names; for each researcher, the titles of their papers; for each paper, a link to the source PDF/HTML document (we simplified this and asked them to just scrape the text of the PDF/HTML link, because ScrapeViz currently does not support scraping links). Note that this data was spread across three levels of pages: a top-level page with a list of researchers; then a page per researcher containing their metadata and a list of their papers; then a page per paper containing metadata including a link to the source PDF/HTML document.

- Yelp: restaurant names; for each restaurant, their hours for today and a list of their most popular dishes. Note that this data was spread across two levels of pages: a top-level page with a list of pizza restaurants in NYC; and a page per restaurant containing their metadata, including their hours and a list of their most popular dishes.

[6]https://web.archive.org/web/20230324030019/https://scholar.google.com/citations?view_op=view_org&hl=en&org=8515235176732148308

[7]https://web.archive.org/web/20230429225251/https://www.yelp.com/search?find_desc=Pizza&find_loc=New+York%2C+NY

Survey and interview: After each task we administered a 7-point Likert scale survey. Finally, after completing both tasks, we conducted an interview to learn about participants' experience using each tool for authoring – in particular asking what they thought of the demonstration and generalization mechanisms, what they thought of each tool's approach for presenting scraped data during (ScrapeViz) versus after (Rousillon) authoring, questions about each tool's specific features, and more general questions about how these tools compared to tools they had used previously for web scraping, and whether they could imagine using either tool for web scraping needs in their life.

5.7 Results

Participants generally appreciated both tools and the utility they offered for reading and authoring scraping macros without needing to write program code. Overall participants appreciated ScrapeViz's visual and interactive nature, enabling them to get an overview of macro behavior, quickly understand the source of scraped data and anomalies, and verify in-progress authoring. Participants also appreciated Rousillon's block-based code and variable names as a description of the scraping logic and data to be collected. Participants suggested that future tools would benefit from a combination of ScrapeViz's visual, interactive interface and Rousillon's block-based program and variable names. They also suggested refinements to streamline ScrapeViz's interface and reduce potential information overload.

5.7.1 Reading

Overall participants' sentiments toward both tools were positive – in the context of the reading task, 9/12 participants agreed or strongly agreed that ScrapeViz was easy to use, and similarly 9/12 participants agreed or strongly agreed that Rousillon was easy to use. The differences between the two tools were more qualitative and needs-specific, as we discuss more below.

All 12 participants were able to correctly identify the specific kinds of data collected for each website. When asked to try to reason about what was causing the anomaly for their given macro, eight participants were able to do so correctly with Rousillon and 11 with ScrapeViz; with a hint (e.g., to consider comparing product or player pages), these numbers increase to 11 for Rousillon. If we look at correctness for understanding anomalies by website, 11 participants were correct for WTA (12 with a hint) and eight for Wayfair (ten with a hint). These results indicate that participants generally experienced more challenges with the Wayfair website and in using Rousillon for understanding anomalies. The Wayfair task seemed to be harder perhaps for two reasons: 1) in Rousillon, the Wayfair anomaly presented as a missing row so it could be challenging for participants to keep track of product pages which were skipped, and 2) all Wayfair product pages in our study did have "variations" listed but sometimes presented different visually; some participants did not realize that a different appearance in data (even though the data still is present) could cause a change in scraping behavior. Participants likely had an easier time with ScrapeViz, because as described more below, ScrapeViz presents an explicit "N/A" when data is missing, while Rousillon presents an incorrect value or excludes the row entirely. I talk more below how this does not allow a perfect comparison between ScrapeViz and Rousillon especially for correctness and completion time metrics, but our interviews with participants still reveal other meaningful qualitative differences between the two tools. Challenges in identifying anomalies and the reasons behind them appears to be slightly skewed toward less experienced programmers; of the five participants who had trouble without a hint, one had no programming experience, one had less than 1 year experience, two had 1–2 years experience, and one had 5–10 years experience.

Tasks using Rousillon took between 7:28 and 16:48 minutes to complete, with a median time of 10:37, while tasks with ScrapeViz took between 1:59 and 12:12 minutes, with a median time of 5:09. We cannot compare these raw times alone though, because these times include the time it took for Rousillon to run to scrape data (which took around 2:30 minutes per website). If we subtract out the time to run Rousillon, the median comparison times for the two tools are 8:37 for Rousillon and 5:09 for ScrapeViz. Participants were quicker with ScrapeViz likely for a couple reasons: 1) participants could see the scraped data immediately with ScrapeViz so could start inspecting it right away (versus with Rousillon data was not already scraped and participants often spent some time trying to understand the block-based code before running the script), 2) as described more below, participants could often align data in the output table with their source pages more easily with ScrapeViz, and 3) again, as mentioned above, for somewhat superficial reasons in how ScrapeViz presented anomalies with an "N/A", it was quicker to identify anomalies. Task completion times on average fall slightly lower for participants with more programming experience, but not consistently.

If we compare participants with 0 to 1 year of programming experience (four participants) with participants with 2 to 10 years of programming experience (six participants), the median task completion times are slightly lower for the more experienced programmers (ScrapeViz: 4:17, Rousillon: 9:14) than for the minimally experienced programmers (ScrapeViz: 6:57, Rousillon: 10:49) for both tools, but their full ranges are not seem very different: experienced programmers (ScrapeViz min: 1:52, max: 7:33; Rousillon min: 7:28, max: 13:04) and minimally experienced programmers (ScrapeViz min: 2:50, max: 7:16; Rousillon min: 9:04, max 12:36). It is important to note that this is a small dataset; an evaluation with more participants would help identify if this pattern generalizes. Additionally, there are multiple possible reasons for why these less experienced programmers took on average slightly more time to complete tasks – it could be due to challenges in understanding computing concepts, in understanding new UIs, or in using under-polished research prototypes containing usability issues or bugs, or a combination of these; it makes sense that more experienced programmers may have more experience addressing these kinds of challenges.

5.7.1.1 Understanding what data is scraped

Participants seemed to find ScrapeViz slightly easier to use for understanding what a macro scraped – 9/12 participants agreed or strongly agreed that it was easy to understand what the macro was scraping using ScrapeViz; for Rousillon, 6/12 agreed or strongly agreed.

One reason participants appreciated ScrapeViz was that they could see scraped data visually highlighted in the context of the website pages – "I think having the highlighting around the specific boxes made it very clear, *this is what it's looking at, this is what it's trying to scrape*" (P5). Presenting multiple website pages with scraped elements highlighted enabled users to discover the pattern of what data was getting scraped – "It's just showing what's being scrapped on the page. I can just look at it and know, *okay, it's scraping this element*, and from a number of examples I can see that for example, *okay, this is the last name of the tennis player, this is their age* and so on" (P3).

By presenting multiple website pages at once, ScrapeViz also helped participants understand the high-level semantic actions performed and the hierarchy of pages visited – "I guess the part that the second tool [ScrapeViz] did better is that they set out the individual pages upfront so that you can see what data is pulling out from and what pages that got scrapped by the tool... you can see all those individual player pages in the first place" (P1). "It was helpful to see how they're grouped together so I can say that this is what I'm iterating through in the first layer, and then for one of them, what else is being iterated through and so on" (P6). In contrast, to understand the exact pages visited with Rousillon, participants either had to carefully watch the pages visited during the linear scraping execution, or after the fact try to align the output table data with website pages and manually navigate – "in the first tool [Rousillon], you could just see one window at a time, so it's not clear how you're navigating between pages as compared to the second one [ScrapeViz]" (P6). "For the first tool [Rousillon], you have to click into the homepage, each furniture, and then you can make sure what data are being scraped" (P1). "I didn't necessarily know, I realized I had to click on their names, but at first I didn't know that" (P7).

Meanwhile, many participants noted how Rousillon's explicit block-based code and variable names were helpful in understanding the scraping logic more concretely – "I can see, *oh, this is a for loop and it's trying to iterate through something*, which the variable names kind of gives a hint and so I can read it and from the variable names I can kind of understand maybe what it might be scraping" (P3). Even some participants with limited programming experience expressed that the block-based program was not too difficult to understand – "I think it's pretty intuitive or easy to understand about the data structure...I learned a little bit coding before, so I know what is...for loop...the language it use is very close to verbal or just like a sentence. So I can know, I don't need to learn coding language, I can just use my normal language sense to know what does it mean" (P11, less than 1 year of programming experience). However, variable names were less helpful when they were not well-named – "it was not as straightforward because it was just showing the script and it was only when I had run it, I knew what `variation` [code variable] was. I was only able to guess based on what are...being shown within the page...I would have no way to tell what `variation` is. Only by running it I would have been able to tell that" (P8).

Some participants leveraged Rousillon's "relevant tables" before running the script, which helped them ground the block-based program and variable names in the context of concrete data extracted at recording time – "I checked the tables, which I guess gives some examples of each what each of the variable is. So from that I can confirm that *yes, this actually means this*. So that makes me understand, *okay, what is it scraping?*" (P3). "I liked seeing the table structure [relevant tables]...I liked seeing that level of detail and found that to be pretty helpful to see *okay, this is what data I'm actually looking to collect* with an example...That I found to be very useful in the first one [Rousillon], whereas the second one [ScrapeViz], it was much more like I don't have any descriptors of what the data is" (P5).

Some people also appreciated the loop indices Rousillon provided in the output table as a way to understand the different items iterated through and the records potentially missing – "I think I liked the output table of the first option [Rousillon] a little bit more because I felt like it gave me, even the numbers, it gave me a sense of *this is a unique row of data*, and I'm very much like a person who works in relational databases a lot, so that makes sense to me...and those numbers to me in my brain translate to a primary key of sorts. It's like, *okay, yep, that's where it came from.* Whereas on the second example [ScrapeViz], just having the data listed out, I didn't get that full context of this is what each item is. So I think, yeah, if you're looking at larger sets where you're trying to scrape multiple things, I think having a little bit more structure around that data is helpful to understand what's being scraped" (P5).

5.7.1.2 Identifying source of data

Participants found clicking on cells in the interactive table highly useful. Firstly, it helped them get a grasp of what each piece of data in the table was, by seeing the source of that data – "Yeah, so I think for the first one [ScrapeViz], I click the table and it navigate me to that part and also highlight it so I can very easily to know the data...I think that feature impressed me a lot" (P11).

Clicking on cells in the interactive table also served a secondary purpose, helping participants navigate as they worked to understand data in the table and understand anomalies – "clicking on the value actually takes you to where that page come from, where that particular data point came from. I think that's great, just being able to investigate, especially if you see something wonky like we did with actually both cases when there were errors" (P9). In contrast, participants noted that with Rousillon, in order to inspect specific pages and potential anomalies, they had to compare the output table with website pages and manually navigate – with ScrapeViz "I don't need to do much effort. I would just go to click the table, click the cell, and as long as I can see the page, it's easy to figure out what's wrong. I think the most effort goes to find the page, and that's where the other tool [Rousillon] doesn't do well is I need to manually find the page myself. I need to check, *okay, this player's name, where is this name in this page one* and find the link. It's all manual, so it's not easy for debugging" (P2).

5.7.1.3 Identifying anomalies

Most participants felt ScrapeViz served them better in identifying and understanding scraping anomalies than Rousillon.

To begin with, there was one superficial difference between how ScrapeViz and Rousillon presented anomalies in the given tasks that likely contributed to some degree in participants favoring ScrapeViz. In the given tasks, ScrapeViz presented anomalies as "N/A"; for the WTA task if no coach element could be found, an N/A was shown in the corresponding cell in the table for that player; for the Wayfair task if furniture variations were not shown as thumbnails (but instead in collapsed panes), a single N/A was shown in the table for that furniture product. Meanwhile, Rousillon showed either incorrect values (a number instead of a coach name for the WTA website) or excluded rows from the table (excluded a full furniture product row for the Wayfair website). It makes sense that participants were often able to discover anomalies more easily with ScrapeViz because they could quickly scan for N/A values within ScrapeViz's table. Meanwhile, for Rousillon, participants had to discover unexpected data types (WTA: a number rather than a name), or had to discover that certain rows were skipped (Wayfair). To discover that certain rows were skipped, participants either had to be paying careful attention during the linear scraping execution to notice certain products were not added to the table; or, they had to notice that the loop indices in the right-most columns of the table skipped certain indices; or, after scraping finished they had to manually compare the table with the website content to notice certain products were not included. Therefore, one takeaway here, though not actually specific to ScrapeViz or Rousillon's larger design, is that skipping items in the output table may not be ideal for helping users uncover scraping anomalies; instead, it is likely better to explicitly include empty values like "N/A".

However, interviews with participants suggest that even if missing data were shown as N/A for both tools, participants still would find benefit with ScrapeViz for navigating and understanding why the wrong data was scraped. With ScrapeViz, once participants noticed an N/A in the output table, they could quickly navigate to the page of interest by clicking on a relevant cell in the table.

Some participants clicked on the N/A cell itself and were directly taken to the website page (e.g., the player's page) and could investigate why the value (e.g., coach name) was not scraped. More participants instead clicked on a semantic identifier in the N/A's table row (e.g., the player's name), which then directed them to the top-level page where that semantic identifier was scraped from (e.g., the list of players on the WTA website); they then clicked on the semantic identifier (e.g., the player's name) on that top-level page, which then directed them to the resulting page (e.g., the player's page) where they could then investigate why a value could not be extracted. It is unclear if participants found this second workflow more intuitive, or if they simply had not had enough experience with the tool to realize they could click on any cell (e.g., an N/A cell) in the table.

Regardless, most participants found clicking on cells in the table or clicking on scraped elements in website pages to be quick and intuitive ways to navigate, for the purpose of understanding anomalies – "you can just click that and it will automatically take you to that page and then you can easily see if that page doesn't actually formed like you think it would and it's actually give you errors. So that's pretty good, especially when errors happening. You can easily see what's causing it by just clicking it" (P3). "I think that it was very much straightforward to figure that out...I just had to click on the particular items. Actually, I could just click on the row in the dataset, it would take me there, which was very much convenient for me, and I do not have to go back to the main page and find that item I was doing initially. And so yeah, I mean, I could figure the anomaly quicker in the second one [ScrapeViz] compared to the first one [Rousillon]" (P4). Some participants did find this navigation to be a bit confusing because of all the many website pages presented in the interface – "I guess the problem with the visual one is that instead of browsing the webpages in your browser, it's actually trying to render the webpage in that small window in a webpage and it's having all the pages that are not active as super small windows and so clicking between them kind of feels unsmooth" (P3).

Alternatively, to navigate in Rousillon, participants had to manually align the output table to website page content and then click to navigate to the desired page. Most participants found this to be tedious or cognitively demanding – "I think in the first tool [Rousillon], I think just the fact that it was opening up a new tab in the browser or you were having to navigate to each page individually, you kind of lost that initial context, whereas I felt like the second option [ScrapeViz], you still retained some of that context, so it was almost like you could see where you were drilling down into the page a little bit easier...I could see the path that I was taking through the webpages a little bit more clearly and could kind of see examples a little bit more easily. So I could click on one tennis player's name, see where that popped up, and go back to the original page, click on another tennis player's name, and it would go to the same kind of spot. So it was very obvious how it was navigating how the tool was working to navigate, whereas with the first one [Rousillon], you lost some of that context. It was almost like the back button on a webpage where it's like, I just want to go back. I don't know what I'm going back to, but I'm going back to something else" (P5). However, some participants preferred this manual navigation in Rousillon, because it was more familiar to them – "I guess one thing good about the Rousillon...is the fact that you can actually investigate the webpage in your browser, which is what you do every day" (P3).

5.7.2 Authoring

Participants had similar sentiments about the two tools in the context of authoring, which makes sense because ScrapeViz and Rousillon are reasonably similar in their demonstration-based authoring. Overall, participants found both ScrapeViz and Rousillon easy to use, with 11/12 participants agreeing or strongly agreeing that ScrapeViz was easy to use, and 10/12 agreeing or strongly agreeing for Rousillon.

12 of 12 participants completed the tasks correctly using ScrapeViz and 11 completed the tasks correctly using Rousillon. The one participant (P2) who did not fully complete the Rousillon task correctly was for the Yelp website, where she correctly provided examples for the restaurant name and hours (one example per each), but then incorrectly provided multiple popular dishes (all of the ones she saw on the page) as examples, which does not follow Rousillon's model of one example per data type. This participant seemed to experience some confusion about Rousillon's generalization model. In section 5.7.2.4 we discuss how this participant and others describe that they found ScrapeViz helpful for getting immediate feedback on their demonstrations, immediately seeing what data is scraped, which helped reduce confusions like this about the generalization model.

Participants took a similar amount of time overall to complete tasks regardless of website or tool, with a median time of 5:53 minutes (min: 4:11, max: 10:49) for Rousillon and 6:13 minutes (min: 3:11, max: 8:42) for ScrapeViz. Note that recording the demonstration sequence in Rousillon was typically quicker than in ScrapeViz. In Rousillon, demonstration and then checking the resulting generated program and results were all distinct steps in the process, whereas in ScrapeViz, generalization and scraping are performed immediately, so users tended to check for correctness along the way. Task completion time also did not meaningfully vary based on participant level of programming experience. Looking at participants with 0 to 1 year of programming experience (4) and participants with 2 to 10 years of experience (5), median task completion times were comparable: minimally experienced programmers (ScrapeViz: 6:44, Rousillon: 5:19) and experienced programmers (ScrapeViz: 6:12, Rousillon: 5:53).

5.7.2.1 Ease of use for less experienced programmers

Even less experienced programmers found authoring intuitive after the training exercise – "I thought it was pretty easy to figure out where the stuff was that I wanted to get, and then just go back and do it once I had figured that out" (P7, less than 1 year programming experience); she explained that the most challenging part was just inspecting the website pages and "[figuring] out the game plan. Like, *oh, I'm trying to get this, but I have to click to get that second piece of the data*". P10 (reported no programming experience) compared the tools to the Data Miner plugin[8], saying "I think these [ScrapeViz and Rousillon] are just a lot more intuitive for someone, especially myself. I mean, I'm not tech illiterate, but I'm certainly programming illiterate. I was able to pick these up pretty quickly, and I think you could, someone who just has basic computer skills could probably be able to learn it fairly quickly. With Data Miner, most of the time, I have to go into the actual HTML elements and poke around to scrape the thing that you want to scrape."

5.7.2.2 Scraping interactions

Some different preferences emerged across participants with regard to specific interactions for selecting. Many participants found ScrapeViz's click interaction to be too sensitive – sometimes they would click on whitespace as they were just trying to navigate around a website page and it would mistakenly trigger that element's container to get selected, which would add a border to it, put its text into the output table, and open a link in a new window. ScrapeViz would likely benefit from some minor design improvements here, perhaps leveraging heuristics to only select an element if the click is within a certain number of pixels of text and not far out in whitespace. As a result, some participants preferred Rousillon's more intentional selection interaction, where they first needed to press the Alt/Option key and then clicking with their mouse would trigger a scrape action. With ScrapeViz, though, participants did appreciate the ability to delete a previous scrape action if they made a mistake or changed their mind, a small feature that Rousillon does not offer.

5.7.2.3 Demonstration and generalization

Overall participants did not care too much about whether they needed to give one example (Rousillon) versus two examples (ScrapeViz). One participant mentioned that giving two examples in ScrapeViz was a bit more work because they needed to come back up the hierarchy tree to generalize each of their initial scrape demonstrations. While this is true, if participants had instead given two examples at once on a single page before adding more scrape actions within their child pages (as described in 5.3.1), that would have reduced navigation needed; in the tutorial video we only

[8]https://daSusanner.io/

taught the approach of coming down the tree to demonstrate and then up the tree to generalize, and did not teach the quicker version with giving two examples right away.

Some participants with programming experience noted that demonstration is quick but offers less control that writing code. Some participants commented that the block-based code in Rousillon is nice in case they might want to edit what was generated from their recording – "this one [Rousillon] seems like a low-code one where it's kind of trying to be a bit more transparent of what's happening and maybe give more knobs and levers for someone to customize what has already recorded" (P8).

5.7.2.4 Real-time scraping during demonstration

A key distinction between Rousillon and ScrapeViz is in *when* they present their scraping results. Rousillon presents scraping results only after the user has stopped recording demonstrations and then run the generated macro. ScrapeViz actually starts scraping immediately as the user provides demonstrations – immediately adding scraped data to the output table and indicating scraped elements within website pages by placing borders around them. ScrapeViz immediately generalizes based on users' demonstrations – generalizing scraping actions on the given page, adding new website pages within the interface, and scraping across those multiple pages in parallel. Many participants found this real-time scraping helpful because it gave them feedback that they were giving the correct number of examples and that their generalization worked as intended – "That's really, really helpful. I think that's a big selling point versus the other tool [Rousillon]. Like I said, especially in this kind of tool, I need to track my own progress. One I have selected and did I actually scrape the thing that I want to scrape and I need to amend myself if I made anything wrong. The first tool [Rousillon] is kind of like a black box. I don't know where I am, what did I do?" (P2). "If I want to get all the authors name and after I select the second one, [ScrapeViz will] highlight all the other authors for me. So I'm pretty sure I did the right thing, but for the second tool [Rousillon], because I cannot get visual feedback after I select the first one, so I would be a little bit worried that I did something wrong or I didn't do it correctly" (P11). "It's just like rather than looking at the script and then going to the webpage to see exactly clicking on the links physically, and then seeing if it's where do you have to navigate versus just having those pages already opened up. As soon as you click on it, it just opens it up for you, and once you sort of repeat it, it generates those pages for all of your elements. So having that is helpful rather than doing it manually in the first one [Rousillon]" (P6). Some participants felt although the real-time scraping in ScrapeViz was helpful for verification, it could potentially be overwhelming to see all data and website pages at once – "I can imagine it being for a page which has few rows, it's okay, but when it becomes a pretty long list of things, it might be hard to track" (P8). "I think I just preferred the first one [Rousillon] where the table would just be there after you'd already set up the scrape. I think it's just a bit cleaner that way because you have all the windows and then also this table that's populating and the table can get quite large. So having to scroll through all that, there's, there's a lot going on with that" (P10).

5.7.3 ScrapeViz – multiple website pages

As mentioned above (section 5.7.1.1), participants found some utility in being able to see multiple website pages at once, for example, getting a sense of parent-child page relationships. Although participants found ScrapeViz easy to use overall, and although it did not seem to impact their performance, many participants did comment that the current interface can at times feel overwhelming with too many website pages presented – "I think it was great having the previous page. Having the next page is great on the screen, but other than that, having 10 other screens on the page is very much distracting. I'd say. I don't want the screens of other items when I'm just navigating a particular sofa or an item that I'm looking at. So for example, if it's a main page, I clicked on an item, it takes me to the next one. So I just want these two screens. I do not want other pages that I can click or I can view. It's just taking up screen space" (P4). Participants suggested showing fewer parallel sibling pages at once, or even showing just one sibling page at a time but indicating that there are more – "Maybe one thing I would change is instead of generating all of them, I would just say that these three are generated and then sort of blank out the rest thing. If you want to look at them, then there's an option where you can expand and look, because I feel like the first three that are generated can give you an idea about what's happening versus having all of them on the screen together" (P6). "Maybe just have the one open that you click on and then maybe have a bubble showing how many other ones there are or something just bubble with a number in it, not have all the windows" (P10).

Some participants also commented on how small the non-active website pages are and how it can be hard to see all of the relevant content. This is in part due to a limitation with Electron [18]; we tried to zoom out for these smaller pages so that users could see more content within the small viewport, but this was not possible because Electron only allows a single zoom level across all embedded website pages at the same domain name.

At least one participant did find it useful to view parallel website pages at the same time. When authoring the Google Scholar macro, after the participant provided demonstrations to scrape paper titles, he scanned the sibling pages to see that correctly they also had borders shown around their paper titles. We had expected more participants to use this feature. There could be a number of reasons why they did not: perhaps the interface is currently too cluttered; maybe the website pages are too small for people to consider viewing their contents when they are small; or maybe viewing multiple sibling pages at once is more of a power-user feature.

5.7.4 Participant use cases

Participants commented that they felt demonstration-based tools like ScrapeViz and Rousillon would have helped them in their past automation or scraping work. P11 said that in her past job her company wanted to collect reviews about their products from the Amazon website and did it manually – "Yeah, so I think it's really useful for me. For now, if this tool is like I can use it, I probably use it for scraping the feedback, I guess because I'm UX designer and sometimes if we need to do some feedback, collect user's feedback on Amazon that say, and then we need to copy, keep copy and paste from Amazon's feedback section and that is painful. So I guess with this tool it can help us to know the product feedback. So I think, and it's very easy to use, cause like, I don't know how to code, so yeah...because I worked before and at that time our products sold on Amazon and we got a lot of feedback and we want to use those feedback to improve our product. At that time, the thing that I did, I just keep copying and paste for thousands of times." "One instance I remember is one of my credit cards gave me options to redeem the points with something, but their navigation was so clunky that I had to actually scrape all the thing to sort it out. And something like this would've just saved, I don't know, 15 minutes of my time where I would've clicked three times and it would've done the whole thing for me. So yeah, definitely a powerful addition" (P8).

5.7.5 Other participant requests

As mentioned above (section 5.7.3), many participants suggested streamlining the ScrapeViz interface to reduce information overload, e.g., by reducing or hiding sibling website pages. Many participants also requested a combination of the two tools: the interactive table, scraped data shown in the context of website pages, and the visual hierarchy of pages visited of ScrapeViz; and the block-based code and variable names of Rousillon.

Although participants found ScrapeViz's interactive data table very helpful for quickly reaching the source of each scraped datum, they did have some other requests to make the table even more digestable. They requested to see labels for each type of scraped data in output table, to make it easier to interpret scraped data upon seeing it for the first time. Relatedly, one participant commented that there is still some disconnect between the data in the table and where exactly it came from within website pages. They are hoping to be able to just glance at the output table and get a sense of where each kind of data comes from (without needing to click and inspect specific cells), perhaps through the same color-coding we used for highlighting scraped data on website pages.

5.7.6 Threats to validity

Like most lab studies, our study enables us to evaluate conditions in a controlled setting, which brings benefits for comparison and choosing scraping tasks that ScrapeViz and Rousillon currently support. However, this comes at the expense of gleaning insights into how people would use ScrapeViz and Rousillon for real needs in their lives. Future researchers and practitioners hoping to build on the ideas of ScrapeViz should observe how users use ScrapeViz in their real lives, to better understand users' distributed hierarchical scraping needs, needs with larger datasets and more complex scraping logic, needs regarding collaboration and maintenance of scraping macros over time as websites change, and to otherwise make usability improvements.

We believe that overall the scraping tasks we chose enabled a reasonably fair comparison between ScrapeViz and Rousillon. However, there are a couple notes to make where comparisons were not perfect.

For the reading tasks, ScrapeViz and Rousillon presented anomalies differently for the websites we chose – ScrapeViz presented N/A for missing values, and Rousillon either included incorrect values (e.g., a number instead of a coach's name) or skipped rows completely (e.g., skipped furniture products when some of their metadata was missing). This is because ScrapeViz and Rousillon have slightly different algorithms for generalizing from user examples; often they'll generalize similarly, but not always exactly the same. As a result, participants found it easier to identify anomalies in ScrapeViz. However, this is not fully representative of how these tools may present anomalies for other tasks, and this is also not the key difference between ScrapeViz and Rousillon that we are trying to evaluate. Regardless, during our interviews we were still able to learn from participants about their experiences with the more defining characteristics of each tool (e.g., how they represented the scraping generalization, the output data's connection to its source webpage, viewing one or multiple website pages at once).

For the authoring tasks, there was one important feature of Rousillon's that was not always available – its color coding of corresponding elements during authoring (e.g., it would highlight all researcher names on Google Scholar with the same color). This feature works on some websites (e.g., our Google Scholar task) but not for many of the websites we considered, and as a result did not appear during our Yelp authoring task. It is possible having this feature may have given Yelp participants more confidence in how their demonstrations would generalize, but this did not seem to have a large impact on performance.

5.8 Limitations

5.8.1 Generalization

As explained in section 5.5.1, ScrapeViz only supports generalization across UI elements whose XPaths can be represented with a formula `/prefix/index/suffix` with numeric variable `index`. Therefore ScrapeViz will treat elements that seemingly look similar in the website UI but do not adhere to a common XPath formula as separate scraping actions. A common failure case here is if two elements visually appear to be parallel (or near-parallel, e.g., one contains some extra whitespace), but one of them turns out to be slightly deeper in the DOM with an XPath formula `/prefix/index/suffix/extraNode`; this may happen if the user selects the innermost DOM element for one item on the page, and for the other item selects a "container" DOM node. Future work could consider heuristics to align selected elements that appear to be parallel even if their XPaths do not follow the exact same formula (e.g., consider parallel if one element XPath can be adjusted to match a formula without compromising text content).

5.8.2 Stateful operations

Currently ScrapeViz only works meaningfully for operations that are non-stateful (e.g., clicking a link that simply navigates to a new page and makes no other alterations; scraping text). If a user performs a stateful operation (e.g., clicking a submit button, clicking a checkbox, deleting an item) ScrapeViz will generalize XPaths as normal, but the actions may not be presented in ScrapeViz's interface in a meaningful way, e.g., clicking a "delete" button will cause that entry to get removed from a list, and if all items are deleted due to the generalization, at the end the list will present as empty; looking at that final empty list, it may be hard for a user to understand what actions were actually performed. Future work may explore visualizations that present intermediate states rather than just the final state of each website page.

Another challenge with generalizing stateful operations is that in the case of an incorrect generalization, a stateful operation could be erroneously performed (e.g., incorrectly deleting an item from a list), which could be detrimental. Future work should explore ways to help users understand generalized stateful operations before they are actually performed. One idea is to provide users a preview, highlighting which UI elements would be clicked without clicking them until the user approves.

5.8.3 Multi-step navigation sequences

Currently the only navigational operations we generalize are single clicks that navigate to a new page. If navigation involves an interaction sequence that opens or closes panes or otherwise navigates within a single page without navigating to a new url, currently we do not generalize this sequence to other items or other pages. That should be relatively straightforward to implement.

A bigger question is how to visualize these generalized sequences of interactions to users. ScrapeViz's interface already presents a lot of information, as participants pointed out, and has limited space to present much more. It would be important to think about the medium to represent these sequences (e.g., a storyboard graphic, a GIF), as well as when to show them (e.g., for all items on a page at once, or only for one at a time).

5.8.4 Identifying different UI states

ScrapeViz intends to convey the actions performed in a web scraping macro and how they generalize across different UI elements and website pages. One aspect of this is to convey the sequence of different UIs visited, grouped semantically (e.g., group parallel actor pages visited for the IMDb example). Therefore, it is important to be able to identify meaningfully different UIs, so we can determine when to continue performing actions in the same website window versus opening up a new website window. Currently we use urls as a proxy for identifying semantically different UIs – if clicking on a UI element (e.g., an actor's name) triggers a new website url to be visited, we render that new url in a child website window and keep the parent website page intact. However, this approach breaks down in a couple cases:

- Single-page apps: In a single-page app, interacting with a UI element may cause the UI to change drastically to a semantically different UI, but this is simply just new HTML served by the server – the url remains the same. With our current approach, the new UI would be rendered in the same website window, but this is a poor representation of the actions being performed and different UIs visited. Ideally this new UI should be presented in a new website window, either as a child window (e.g., clicking on an actor name on a movie page in a single-page app version of IMDb) or a sibling window (e.g., clicking on a parallel tab that presents the same kind of information but different data)

- Elements with urls that should be considered siblings and not children: In most of the examples we have discussed in this work, clicking on a UI element link (e.g., an actor name on a movie's page) results in visiting a child website page (e.g., the actor's page). However, it is sometimes the case that clicking on a UI element link will result in visiting a semantically *parallel* UI to the current one. For example, clicking on a different heading on a restaurant menu website may take you from the lunch menu to the dinner menu, each at their different urls. Even when clicking the dinner menu heading when on the lunch menu page, this should ideally render the dinner menu page as a parallel page to the lunch menu page, not a child page.

In both of these cases, it will be important to explore better approaches for identifying UI types – whether two UIs are the same, are the same type but present different data, or are completely different. Rather than considering urls alone, future work may consider heuristics, or language or vision [66] based machine learning approaches for comparing the textual content, structure, and/or appearance of UIs.

5.8.5 Hierarchically grouping same-page scrape actions

Currently ScrapeViz's understanding of hierarchical actions is based on the hierarchy of website pages visited. ScrapeViz does not have any understanding of potential semantic relationships between different scraping actions on the same page; e.g., if a user were to scrape a movie title and its genre and duration from the top-level IMDb page containing a list of movies, ScrapeViz would not actually understand that the genre and duration are for that particular movie, and would simply treat them as separate actions. This has implications for both output presentation and authoring. First, currently ScrapeViz would just present the above movie title, genre, and duration as three separate rows (single column) in the output data; if the user generalizes each action to collect all movie titles, all genres, and all durations, each would appear in their own single-column row, which very likely is not what the user would want. Second, if the user wants to scrape the rest of the movie titles, genres, and durations on the page, currently they will need to generalize each of these scraping actions separately, providing a second movie title, second genre, and second duration. That seems potentially tedious. Future work should explore ways to either automatically infer semantic relationships amongst separate scrape actions on a single page (e.g., like Rousillon does), or provide the user an interaction to do so themselves.

5.8.6 Website viewports

Initially we hoped to present small and large website viewports at different zoom levels, with small viewports zoomed out so users could see more of their contents (a request participants also had in our study). However, currently ScrapeViz presents small and large website viewports at the same zoom level due to a limitation with Electron [18], where all website windows of the same domain name are presented at the same zoom level. Future work may want to explore alternative solutions. One idea may be to show zoomed out screenshots for small website viewports rather than live pages, since users do not actually interact with the contents of small viewports while they are small anyway. Then when the user selects a small viewport to make it large, ScrapeViz could swap in the live web page.

5.9 Discussion and Future Work

5.9.1 Understanding web scraping macros

Overall participants found both ScrapeViz and Rousillon easy to use, and they appreciated aspects of both tools for understanding distributed hierarchical web scraping macros.

In particular, participants appreciated how ScrapeViz helped them understand scraped data in the context of the source website page – both through the interactive table taking them directly to a datum's source, and through seeing scraped data visually highlighted on a given website page. This helped them understand concretely what data was scraped, rather than needing to manually align an output table with many possible source web pages. Participants found the interactive table especially helpful for navigating across different pages to understand the reasons for anomalies.

During authoring, many participants found the immediate feedback from ScrapeViz helpful, to understand whether they were giving the correct number of scraping examples and that their generalization worked as intended, rather than needing to wait until they finished "recording" as in Rousillon.

Participants appreciated high-level overview features that both ScrapeViz and Rousillon provided – ScrapeViz's visualization of the sequence and groups of different pages visited (and the data scraped) and Rousillon's block-based code and variable names. Each offered their own unique benefits. ScrapeViz's overview was more example-based, helping users concretely understand the hierarchy of pages visited. Rousillon's overview was more abstract, helping users understand the macro's logic specifically. Participants had different preferences between the two, with more experienced programmers appreciating Rousillon's block-based code for expressing the truth about control flow (e.g., looping, and actions on each page).

Participants' differing preferences around scraping macro representations brings up interesting questions about what is the ideal way to present macros that is appropriately descriptive but still digestable to novices. Likely we should continue to work toward some "ideal" that is easier for everyone to understand, but at the same time, I suspect the solution is more a spectrum of possible tools that users can choose from and float between as appropriate based on their background and current needs. For instance, Webflow [42] is a low-code tool for building websites. Users start out with a purely no-code tool where they can drag and drop to place UI elements on a canvas and use default styles and behaviors. As users need more customization and as their skills increase, they can begin using Webflow's tools that are still low-code but may require more of a computing background, e.g., interactivity, responsive UIs, and live data from data sources. Finally, for customization not supported through Webflow's low-code tools, users can edit source code directly. I believe future web scraping tools should be on a spectrum like this, perhaps ranging from purely no-code AI-powered tools, to tools that enable users to read or edit small snippets of element selection or automation logic, to full-on code editors. Perhaps ideally it should be in a single environment to support graceful floating from one part of the spectrum to another as users needs and skills change, or as they collaborate with others with different skillsets. A challenge, though, would be how to express logic written in code in a no-code or low-code environment whose abstractions cannot express everything that is written in code.

5.9.2 Low-code authoring

5.9.2.1 Demonstration interaction model

In ScrapeViz, users provide exactly two examples of each kind of UI element they want to select, and then ScrapeViz infers generalized logic to scrape the rest. We instead may have considered inferring from a single element example (like Rousillon [55]) to reduce user effort or alternatively more than two examples to support more nuanced inference. The novel contribution of our work is not in the demonstration interaction model or inference technique but instead in the visual representation to support understanding distributed hierarchical web scraping macros. As a result we chose a two-example demonstration model which enables a simple inference algorithm that is reliable in enough scenarios for us to to illustrate and evaluate our novel visual representation. As we discuss more below, future work should explore additional interaction models and inference techniques to support a wider range of web scraping logic and make authoring even more intuitive.

5.9.2.2 Correcting inferences

Currently ScrapeViz only supports limited editing – the ability to add or remove actions and the ability to add a generalization based on two examples. However, PBD web scraping tools should also support the ability to correct or further specify desired inferences, for example by providing additional positive or negative scraping examples, or by specifying required characteristics through natural language as in SemanticOn [106].

5.9.2.3 Leveraging more advanced AI

As explained in section 5.8.1, ScrapeViz currently can only identify two elements as being parallel if they share a common single variable index-based XPath formula. This was sufficient for our purposes of exploring a novel visualization and context tool, but is too limited for real-world tools. Ideally ScrapeViz should consider other more semantic factors in understanding UI element similarity – from small tweaks to what DOM features we consider in an XPath (e.g., CSS classes), to factors beyond the DOM like natural language and visual appearance. Such changes would improve performance for our current two-demonstration interaction.

At the same time, future work should also explore supporting natural language instruction using large language models (LLMs). SemanticOn [106] has taken a first step in this direction, allowing users to augment their scraping demonstrations with a natural language description of why they have selected a particular item and their required filtering characteristics. Recent commercial tools MultiOn [26] and Adept [3] seek to support users in automating web tasks through natural language alone. These projects are very exciting, but it will also be important to understand their possible failure cases and potential human-AI breakdowns. When the user provides a natural language (NL) request, will the AI be able to understand the necessary order of those actions and what pages each part of an NL request should take place on? What is the right balance between speed (i.e., the AI automatically performing all steps) versus asking the user for confirmation after each step? When the AI does something wrong, can the user understand why and how to instruct the AI to do the correct thing the next time? Likely it will make sense to allow the user to use a combination of natural language instruction plus pointing or demonstration to ground their language to concrete elements on the page. These are just a few of the many questions that will need to be addressed to make web automation possible through natural language requests. Additionally, future systems will need to be able to determine what operations are stateful and which stateful operations are easily undo-able and appropriate to automate (e.g., adding an item to your cart) and which need a user's explicit approval (e.g., making a purchase, permanently deleting something).

Specifically to support web scraping, many people have found ChatGPT's code interpreter to be useful in scraping data from a single website page [48]. The user downloads an HTML file from the target website and feeds it to ChatGPT with a prompt of what kind of data to scrape, and optionally examples of HTML elements to scrape from; behind the scenes, ChatGPT generates code that parses the website page and places the scraped data into a CSV file for the user. This is very exciting but is bound to fail in some cases where it scrapes the wrong data or formats it incorrectly in the output CSV file. Currently users seem to be using trial and error with their ChatGPT prompts or searching the web for tips on how to get it to do exactly what they want. It would be more ideal, though, if ChatGPT exhibited more of a dialogue with the user to ask for clarification or share multiple options with the user when there is an ambiguity. To build an LLM-powered distributed hierarchical web scraping tool that supports natural language requests, it would be important to consider many of the same questions as I posed above for MultiOn and Adept. I think a visual macro representation like ScrapeViz will prove especially useful so users can understand the sequence and groups of pages visited and whether scraping works as intended across various pages, especially if the LLM-powered tool were to generate large portions of the macro at a time.

5.9.3 Checking for correctness across dozens or hundreds of webpages

During our study we observed that participants checked only a few sibling pages before announcing their understanding of scraped data or anomalies and moving on. One likely reason they did not check more (or all) of the pages is that it was a user study task, we did not incentivize them to check all pages, and they would not need to use this macro for important work in their own lives. However, even for users using a tool like ScrapeViz for real world tasks, it is unlikely they would have the time to check all sibling pages, and especially not if the number is in the hundreds or more. At the same time, users likely want to develop some degree of confidence in the correctness of the results without needing to check every page.

This suggests additional work is needed to help users more quickly uncover different patterns that may arise across pages. One idea is to cluster pages by their visual or DOM similarity and/or similarity in their scraping results. For example, for the WTA women's tennis example from the user study, ScrapeViz correctly scraped the player's age and coach's name for most players, but for some players an anomaly occurred – no coach was scraped because there was no coach listed on the page. In this case, it likely would have been useful to users to see two distinct clusters of pages – 1) player pages that include a coach's name, and 2) player pages that are missing a coach's name – to more quickly identify and understand the anomaly.

5.9.4 Adapting for blind and visually impaired users

ScrapeViz is a primarily visual system and would need extra work to effectively support blind or visually impaired users through a screenreader. Supporting demonstrations on an individual website page should be relatively straightforward, as users can use screenreaders' currently capabilities to navigate through the website DOM and interact with UI elements. However, providing blind and visually impaired users an equivalent understanding of macro behavior across website pages would require some more significant work. Likely we would want to offer easily navigable information structures with references to website DOMs, i.e., the tree structure of actions performed and pages visited, lists of sibling pages, and lists of sibling UI elements. After navigating to a desired page, users could then use their screenreaders as they would normally to parse that individual website page or a UI element on it. This alone still probably does not enable screenreader users to efficiently interpret website pages and macro behavior. Sighted users have the benefit of being able to glance at website pages to quickly interpret their content and make assessments about macro behavior. We need to offer blind and visually impaired users corresponding “glanceability” capabilities – some ideas include auditory or written summaries of page content, comparisons of pages, comparisons of UI elements, and descriptions of surrounding UI elements, all perhaps generated leveraging LLMs. Any future efforts to make ScrapeViz accessible should be co-designed with blind and visually impaired users to ensure we meet their needs.

5.9.5 Internet browsing and tracking pages visited

We specifically designed ScrapeViz for authoring and understanding *web scraping macros*, but its multi-viewport design may lend itself well to other internet browsing activities, too. Someone planning to purchase a computer monitor may want to browse different options online, within one or multiple stores’ websites. A tool like ScrapeViz may help them keep track of the different pages they have visited or that they plan to visit and be able to observe them from a birds’ eye view, e.g., grouped by store website or subcategory. Users could highlight relevant information on each page, either manually per page or using a generalization feature like in ScrapeViz, e.g., to highlight the rating for each monitor. Different from ScrapeViz’s current model, perhaps users could have more flexibility in what pages they keep and how they organize them; maybe they could remove product pages they are no longer interested in and filter or group pages more semantically based on their characteristics (e.g., speakers or no speakers, adjustable height, resolution) across different websites. Such a tool could help users during their own browsing process, as well as serve as an artifact for collaboration.

CHAPTER 6

Discussion

In this thesis I have studied and designed tools for web automation. My work provides a step forward for helping users more easily author and understand web automation macros. At the same time, there are limitations to my work and opportunities for future work, which I describe below.

6.1 Contributions

6.1.1 Intellectual

My thesis contributes new insights about how programmers write web automation code and the challenges they face, and novel user interactions and inference techniques for authoring and understanding web automation macros. First, I studied the challenges of programmers writing web automation scripts and found that a key challenge is in interpreting a website's Document Object Model (DOM) and identifying correct and robust element selection logic (chapter 3). Next, I built two programming-by-demonstration (PBD) tools that enable users to author web automation without writing code: ParamMacros (chapter 4) and ScrapeViz (chapter 5). Both tools leverage structural patterns within a website's DOM to identify likely elements for generalization. My work also contributes tools that help users understand web automation macros – ScrapeViz (chapter 5) and my prototype web automation IDE (chapter 3), through live website viewports, provide users context for what effect their macro has on the target UI and what elements are scraped. ScrapeViz additionally presents these website viewports in a storyboard-like visualization to provide a glanceable visual, and enables users to understand scraped data in the context of their source website pages.

6.1.2 Scholarship

My thesis contributes to existing bodies of work on developer and programming by demonstration tools for web automation. We have published chapters 3 (developer challenges)[1] and 4 (ParamMacros)[2] at conferences. Chapter 5 (ScrapeViz) is recently completed work that we plan to submit for review at an upcoming conference.

6.1.2.1 Developer challenges and needs in web automation

Many tools such as Selenium [35], Puppeteer [31], Playwright [29], Cypress [14], and Beautiful Soup [8] exist to support developers who write test automation or web automation code, but previously the literature lacked a description of how developers use these tools and the challenges they face. My work in chapter 3 contributes an evidence-backed description of the challenges and needs of programmers writing web automation code. Challenges include selecting UI elements, generalizing scripts to work across different inputs and pages, keeping track of nodes in the DOM, handling navigation and timing, dealing with an at times long feedback loop, and potential future website changes breaking scripts. Based on our observations, I provide recommendations for practitioners and researchers to consider when designing future web automation developer tools – namely, (1) that they include contextual information about the target website page (e.g., UI snapshots, DOM nodes and values) within the programming environment to bridge the "gulf of execution" [102], and (2) that they offer effective feedback (e.g., on selectors, interactivity of elements, across pages, and over time) to bridge the "gulf of evaluation" [102].

6.1.2.2 PBD for web automation

Prior work in record-and-replay tools and programming by demonstration (PBD) have explored approaches to make creating web automation macros easier. Record-and-replay test automation tools such as Selenium IDE [36], iMacros [22], and Cypress Studio [15] provide a start, enabling users to record a single UI interaction trace which is automatically converted to a script that they can replay for testing purposes; this is fairly limited, though, for web automation, where users often want their macros to work for different user inputs or website page variations. CoScripter [95, 87] was an early PBD tool for web automation that enabled some more flexibility in its replay – users could record a UI interaction trace and the system would generate a "sloppy program"

[1]Rebecca Krosnick and Steve Oney. Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2021).*

[2]Rebecca Krosnick and Steve Oney. ParamMacros: Creating UI Automation Leveraging End-User Natural Language Parameterization. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2022).*

of natural language commands, which would be interpreted and run in the context of the website page, allowing some flexibility in case of future page changes. CoScripter enables some degree of generalization, allowing users to replace data such names or email addresses with variables so that they can be easily reused by colleagues, but this is limited to dynamic form-filling and does not support dynamic element selection.

My work in chapters 4 (ParamMacros) and 5 (ScrapeViz) introduces new PBD user interactions and inference techniques to enable dynamic element selection. With ParamMacros, users are able to create parameter-based element selection automation by providing a natural language request, a demonstration of user interactions of how to fulfill that request, and parameters to describe possible variations of that natural language request; the system then aligns structural patterns in the website DOM with the user-provided demonstration and parameters to infer a generalized automation macro. Most related to ParamMacros is the PBD system Sugilite [88] which supports similar parameter-based automation and leverages similar techniques for inferring parallel UI elements from tree-based view hierarchies. The key differences are relatively minor – with ParamMacros users parameterize their natural language request themselves while Sugilite automatically infers parameters through keyword matching of natural language to text labels in the UI. There are pros and cons to each approach, but likely a combination of the two would be ideal – saving users effort with Sugilite's automatic parameter inference, while allowing them to make corrections or clarify ambiguities using ParamMacros's manual approach. Another difference between the two tools is that ParamMacros is built for *web* automation whereas Sugilite is built for *Android app* automation, though they leverage similar techniques for inferring UI element patterns in the website DOM and Android view hierarchy. Follow-on work [90, 91] to Sugilite also enables more advanced inference and macro creation than ParamMacros. Appinite [90] leverages natural language understanding to better align a user's natural language request to semantic and spatial relationships in the given UI. Pumice [91] enables users to describe ambiguous concepts and encode conditional logic in their programs. AutoVCI [104] and VASTA [110] also support parameterized automation similar to Sugilite and ParamMacros, with some differences. AutoVCI asks the user a sequence of strategic yes/no questions to help clarify the appropriate app, actions, and parameters to automate. VASTA uses computer vision to identify from a UI screenshot the appropriate UI elements to interact with, instead of programmatically interacting with the UI's view implementation. Table 6.1 summarizes how ParamMacros compares to these other PBD tools for parameter-based macros across different axes. The automation scenarios I address and inference approaches I leverage in ParamMacros may not be entirely new, but I offer evidence of their promise for web technologies and show that involving humans in identifying parameters could be a useful tool.

With my system ScrapeViz, users are able to create macros that scrape hierarchical data that is distributed across different website pages. Users interact with website pages and provide two ex-

PBD for parameter-based macros							
Characteristic	CoScripter [87]	Sugilite [88]	Appinite [90]	Pumice [91]	**ParamMacros**	AutoVCI [104]	VASTA [110]
UI type	Website	Android app	Android app	Android app	Website	Android app	Android app
Supports parameter-based text entry	Yes	Yes	Yes	Yes	No	Yes	Yes
Supports parameter-based element selection	No	Yes	Yes	Yes	Yes	Yes	Yes
Leverages structural patterns in UI to generalize	No	Yes, Android view hierarchy	Yes, Android view hierarchy	Yes, Android view hierarchy	**Yes, website DOM**	No, instead leverages list-based heuristics and vector similarity metrics	No, instead leverages computer vision
Parameter identification	User-driven (values they enter in datastore)	Automatic by system	Automatic by system	Automatic by system	**User identifies parameters in natural language input, and system assists suggesting alternative values**	Automatic by system	Automatic by system
End-user interacts with a virtual assistant	No	Yes	Yes	Yes	No	Yes	Yes
Leverages natural language understanding for interpreting request in context of UI	No	No (only simple keyword-based matching)	Yes	Yes	No (only simple keyword-based matching)	Yes, vector similarity metrics and simple keyword-based matching	No (only simple keyword-based matching)
Supports conditional logic	No	No	No	Yes	No	No	No

Table 6.1: Comparison of the authoring characteristics that different PBD tools for parameter-based macros offer

amples of a given kind of UI element they would like to click or scrape, and then the system infers which other similar elements on the page to select. As the user interacts, inferred UI elements are highlighted and new website pages visited open in new viewports, resulting in a multi-viewport view. ScrapeViz leverages a similar inference technique to ParamMacros, leveraging DOM structure similarities to generalize to other UI elements. Most related to ScrapeViz are Rousillon [55] and WebRobot [64] which are other PBD systems that similarly enable distributed hierarchical web scraping. With Rousillon, the user provides a single example of each kind of desired UI element and then Rousillon automatically infers other parallel elements on the page to select. With WebRobot, the user begins manually scraping UI elements and then as the system detects a pattern (i.e., after two parallel elements) it suggests the next element selection action to the user. Rousillon requires the fewest number of examples from the user (only one example per element type compared to ScrapeViz and WebRobot's two examples) which likely results in lower effort when inference is correct, but may result in less accurate inference more often due to higher ambiguity of user intent. Regardless, inference will not be perfect with any of these three systems and it is important that systems like these allow the user to correct inferences and/or provide additional clarification up front. ScrapeViz currently does not support correction. Rousillon supports limited editing to its inferred table relations and to its block-based program script. WebRobot allows the user to decline suggested actions. SemanticOn [106] builds on WebRobot and allows users to specify semantic conditions for what data should be scraped, providing users some additional control. Future work should continue to explore intuitive interactions for users to specify their intent and refine inferences in ways that that systems are able to understand and incorporate.

ScrapeViz's multi-page view also uniquely offers users a broad and immediate understanding of how their demonstrations generalize and what elements will be scraped across different website pages. With Rousillon and WebRobot, this feedback is not as immediate and macro behavior can only truly be verified by running it linearly. Rousillon users can only verify how their demonstrations generalize after they have finished recording, at which point Rousillon will generate a generalized macro. Users can then review Rousillon's block-based script and relations tables, but ultimately will want to run the macro to verify behavior visually. After inferring a generalization, WebRobot shows users the next item it will select, and each subsequent item one at a time as the user accepts them. Alternatively, with ScrapeViz as users provide demonstrations and the system generalizes, users can see the whole sequence of selected UI elements at once. In our user study of ScrapeViz, participants found it especially helpful to see these generalizations immediately to feel confident that they provided the right demonstrations to achieve their desired behavior. MIWA [57] is recent work that builds on WebRobot and provides users a semantic understanding of their macro through natural language descriptions of each program step – users can also hover over a given step which will highlight all corresponding UI elements on the target website page. MIWA shares simi-

PBD for web scraping					
Characteristic	Rousillon [55]	WebRobot [64]	**ScrapeViz**	SemanticOn [106]	MIWA [57]
Support distributed hierarchical web scraping	Yes	Yes	Yes	Yes	Yes
# of examples required for generalization	1	2	2	2	2
Natural language specification	No	No	No	Yes	No
Real-time generalization feedback	No	Only next item	**Across all elements and multiple pages**	Only next item	Across all elements
Code	Block-based	No	No	No	No
Natural language description of macro	No	No	No	Only for semantic conditions	Yes
Element relations revealed	Yes (“relevant tables”)	No	No	No	No

Table 6.2: Comparison of the authoring and understanding characteristics that different PBD web scraping tools offer

larities to ScrapeViz in highlighting all UI elements inferred from a user’s demonstrations at once, but only does this for a single website page at a time – ScrapeViz enables users to see element selection generalizations across parallel website pages. Table 6.2 summarizes how ScrapeViz compares to these other PBD tools for web scraping across different axes.

6.1.2.3 Visual representations and context for web automation

As just described, ScrapeViz provides users novel visual representations for understanding web scraping behavior across different website pages. I believe visual representations and UI context like this likely would be valuable in any context where a user is creating or verifying web automation, whether that is using PBD, writing code by hand, or interacting with an AI agent. For example, my work in chapter 3 shows that tools like Cypress and my prototype web automation IDE (which show target UIs side-by-side with code, the effect a given line of code has on a target UI, and selected elements highlighted) are helpful for programmers. As web automation AI agents like Adept [3] and MultiOn [26] develop, it will be interesting to see how visual representation and context features like those in my systems may become helpful. I suspect they could be especially helpful as descriptions of the steps the AI agent has performed, the logic and reasons behind them, and an interface for the user to clarify or correct.

6.1.3 Engineering

This thesis produces three research prototypes: a web automation IDE for developers (chapter 3), ParamMacros (chapter 4), and ScrapeViz (chapter 5). These prototypes were largely built for exploration and experimentation and are not available for public use, but I make public demo videos for each prototype.[3]

6.2 Limitations of my studies and systems

6.2.1 Non-programmers

In my work I have designed two PBD tools for web automation, with the vision to support novice or non-programmers in creating web automation macros. In my studies I have recruited some participants with zero or minimal programming experience, and they have been generally successful with the tools, but all have been tech literate and therefore do not represent the full range of potential non-programmer users. My user study results should be interpreted accordingly. Future work should more specifically focus on the challenges and needs of non-programmers when designing PBD systems, likely through co-design with users. Although my PBD systems do not require users to write or read code, there still may be concepts that are challenging for users without a computing background, e.g., the concept of generalization.

6.2.2 Usability and learnability

The results of my user studies must be interpreted knowing that participants were provided training on the tools. Although my work focuses on intuitive interaction models for creating web automation, the prototypes I have built do not necessarily meet usability standards for production software and would likely need refinement. For example, as I discussed in chapter 5, some user study participants found ScrapeViz to be cluttered and sometimes overwhelming due to the many website viewports and suggested showing fewer websites at a time. Additionally, in our user studies we presented detailed, 5–10 minute video tutorials to participants and allowed them to ask questions as they tried out the tools – without such training, they likely would have encountered blocking

[3]Web automation IDE: https://dx.doi.org/10.7302/21954
ParamMacros: https://dx.doi.org/10.7302/21953
ScrapeViz: https://dx.doi.org/10.7302/21952

challenges impeding their ability to complete tasks. It is very important that production software used for real-world tasks be better tested for usability and learnability, but this was not the focus of my work. The goal of my work was not to produce perfect usability but instead to explore novel interactions for how users may author and understand web automation macros, and then evaluate their promise through user studies: whether with some teaching people found them understandable and effective, how users interacted with these tools, and where users may find these tools useful in their own lives. After seeing the promise of my proposed interactions, other researchers or practitioners may now build on these ideas, and if releasing commercial products they should focus on user testing to improve usability and learnability.

6.2.3 Real-world use

My studies only evaluated how participants used my tools in a lab setting. This provides an initial understanding of usability and usefulness, but we do not know how users would use these tools for their real-world tasks and over time, and the associated challenges they would encounter. Some prior work has studied how people use web automation PBD tools in collaborative and work environments [87, 74]. In addition I believe it would valuable to study the challenges people face using these tools as websites change over time (which could cause automation macros to break, see section 6.5 below) and as automation needs evolve.

6.3 Web automation and AI

6.3.1 Limitations of DOM structure-based inference

In ParamMacros (chapter 4) and ScrapeViz (chapter 5) I used a simple DOM structure-based heuristic for identifying parallel UI elements on a website page. This was sufficient for exploring initial ideas in parameterized macros and visualizing distributed hierarchical scraping macros, but not appropriately robust for real-world use. When testing out both tools I saw many examples where this structure-based heuristic could not encode my desired automation logic: edge cases where one item in the list has extra content (e.g., a badge) or an extra container around it, resulting in that item not being appropriately scraped or clicked; rows or entries of data that are separated by content like an ad, grouped alphabetically, or otherwise grouped within separate divs, resulting in scraping ending prematurely; items that ideally could be considered values for a parameter in ParamMacros but lack structural similarity or labels (e.g., duration, genre, and director for movie titles on the IMDb website).

6.3.2 Leverage an ensemble of AI methods

We could make tweaks to our DOM structure-based inference algorithms to support some of the above scenarios, for example considering CSS classes and attributes more heavily. Inevitably, though, this will not cover all scenarios, and especially not those where the DOM lacks CSS semantics that align with the user's intent. Some user element selection goals may be better addressed by considering natural language and visual appearance of UI elements on the page. Therefore, future PBD web scraping systems should leverage a combination of approaches for interpreting a website page – in addition to structural patterns, also natural language processing and computer vision [110]. It will be important to consider how to effectively combine these different inference approaches – prior work in Appinite [90], DiLogics [107], and WebGUM [69] provide a starting point.

6.3.3 LLMs and web agents

Beyond just improving inference for our demonstration-based interaction model, there are other interactions that are likely now possible given the recent growth in access to large language models (LLMs). Projects like MultiOn [26] and Adept [3] are exploring enabling end-users to automate interactions with their web browser simply through natural language commands, potentially reducing the effort needed to automate. These will be quite powerful, but it will be very important to explore effective human-AI interaction. If the AI has low confidence in some aspect of the user's request, it should ask for additional information. If the AI detects an ambiguity in the user's request in the context of the website page, it should ask the user for clarification and potentially present the user with options to choose from. There should be a balance between speed (e.g., the AI automatically performing multiple operations in a row) and ability for the user to step in and understand or correct. Certain stateful operations likely should never be performed by the AI and should instead be left to the user, e.g., clicking a button to make a purchase. The user should also have full control to act or proactively provide context. If the user becomes frustrated with the natural language interface or simply thinks it will just be easier to interact with the website themselves, they should be able to. For example, they could fill in a specific step the AI is having trouble with; or they could provide examples and a natural language explanation describing the data they do and do not want to scrape. I think this points to the benefit of a mixed-initiative solution and supporting multiple kinds of interactions. I believe this also suggests demonstration-based approaches still have their place and will not necessarily be fully replaced with natural language interfaces powered by LLMs.

It can be tempting to imagine that users will simply be able to provide a natural language web automation request and the AI will generate code for it or perform it. A challenge is that users need to understand what this AI-generated automation will actually do and if it matches their goal. If a user is using ChatGPT to generate some web automation code, to try to understand what it does they will likely do some combination of inspecting the code and actually running it. However, it can be challenging to understand large chunks of code written or generated by another entity. I believe challenges in understanding LLM-generated programs speak to the importance of the macro representations and UI context work I contribute through ScrapeViz (chapter 5) and my web automation IDE (chapter 3). These will help users understand the high-level steps of a web automation program and specifically what each of those steps does in the context of the target website page.

6.4 Customizability vs ease of use

Participants in our ParamMacros and ScrapeViz studies appreciated how quick and easy it was to author automation macros via demonstration. Participants with more programming experience noted, though, that no-code or low-code tools have their tradeoffs – they are easy to use but offer less control than actually writing code. To be simple to use, no-code and low-code tools inherently abstract away lower-level logic (so that users do not need to write it or necessarily fully understand it) and work within a limited, often task-specific scope. This may be fine for users whose tasks fall within the specific scope the no-code/low-code tool designers designed for, but will be a problem once users' tasks go outside that scope.

One view, albeit a more traditional one, is that arbitrary customizability truly is only possible by handwriting code. No-code and low-code tools like PBD are useful for well-defined relatively common tasks and parameters. For tasks or parameters that are less common and not able to be expressed through a given no- or low-code tool's abstractions, people will need to move to lower-level abstractions, e.g., writing something more code-like. One potential direction for future work is to design environments that support seamless transitions between higher and lower levels of abstractions, so that users can start with no-code for a given task, and then transition to low-code or code as appropriate based on their needs and skill level. Offering users single environments that support multiple levels of abstraction could be highly useful for helping users transition more seamlessly. It could also be useful for collaborative efforts where different team members have different skill levels and different needs (e.g., for a given project some people may only need to see the bigger picture and do not need to see or understand the details, so a higher-level of abstraction is fine). A challenge though is how to express lower-level customizations in a higher-level environment. It will not be possible to express these lower-level customizations purely through the higher-level abstraction, but perhaps they can be conveyed through a natural language note (e.g., generated by an LLM) that conveys the semantics without precision.

An alternative more optimistic view is that perhaps LLMs, which support text generation for a wide breadth of contexts, will help us toward a no-code/low-code future that supports arbitrary customizability. For a goal that cannot be expressed through the abstractions of a no-code/low-code tool, perhaps the user could simply describe their goal in natural language (maybe in combination with direct manipulation) and then the system could use an LLM to make changes to the underlying web automation code. This would be very promising, but there are some very important challenges to address. First, it can be risky to allow an agent to edit code directly. Guardrails should be put in place to ensure the new LLM-generated code meets certain requirements (e.g., leveraging certain APIs, structuring data in a certain way) and does not cause any harm (e.g., delete code, delete data from a database, share private information). Second, even if the LLM-generated code is safe and meets certain requirements, it still may have bugs or not behave exactly as the user intends. The user may need to make additional corrections or edits, but this is incredibly hard to do successfully without a correct mental model of the underlying program. This points to the importance of future work in human-AI interaction, especially for AI to effectively communicate its actions and the reasons for its actions to users. Perhaps the AI could provide a natural language description of the semantic changes it made, but this of course will lack precision, so it could still be very tough for the user to develop an accurate mental model of the program. While researchers work toward bridging the human-AI gap, in the meantime I believe contextual, general purpose tools like some I have developed in my thesis work will be valuable for understanding and debugging no-code/low-code automation programs – for example, tools that visualize execution steps, elements interacted with, or changes in program execution or websites over time.

6.5 Website changes and macro repair

Website content, designs, and implementations change over time, which is very likely to break previously created web scraping macros (as we briefly observed in chapter 3). It could either change the behavior of an existing macro (e.g., a different element is selected now or a different website page visited) or completely break the macro (e.g., the expected element no longer exists on the page). Breaking changes like this are especially likely for ParamMacros and ScrapeViz given our simplified structure-based inference. Even small structural changes that do not really change how a website looks could break our index-based XPath formulas, e.g., if an ad or a new content pane is inserted at the top of the page, this may change the target element's index-based XPath within the page.

Future tools should support users through such likely breakages. One approach would be through smarter, more semantic inference that leverages other clues besides just structural patterns, e.g., NLP and computer vision.

Relatedly, future tools should support some kind of automated repair. The web scraping environment could proactively discover when a macro's behavior changes and then either automatically repair or propose potential repairs to the user. Prior work in repairing UI regression tests leverages computer vision and intelligence about code changes for identifying changes in the UI under test and proposing a new corresponding target UI element [113, 105, 124]. Stocco et al. [113] even account for changes to what website page a target element appears on, crawling the web app to determine if the target element has been moved to another page. These AI systems should work in concert with users, explaining their reason for a given repair and requesting help or accepting corrections from the user. For example, if the system has low confidence in a a repair to propose, it could ask the user to point out the correct new corresponding element to use or to record a new demonstration sequence.

Finally, even if future web scraping tools attempt automated repair, they should still give users all resources possible to reason on their own. For example, users would likely benefit from seeing their macro's execution on the target website as it changes over time. At each point in the macro's action sequence, it could show the user diffs between the website versions (in the DOM, client-side JavaScript, and visually) so they can try to assess why the macro behavior has changed or broken.

6.6 Web automation vs UI automation

This thesis focuses on tools for authoring and understanding web automation macros. Many of the technologies enabling my techniques are web-specific, e.g., constructing XPath formulas for selecting parallel UI elements on a website page, using Electron [18] to render multiple live web pages at a time through WebViews. However, conceptually I expect my contributions would generalize to automation of any kind of digital UI, e.g., mobile UIs, smart TVs, car screens, coffee machine screen. Regardless of UI type, demonstration-based interactions for authoring and visual representations and UI context for understanding should look similar. Perhaps for certain screen types, like a car screen, demonstration by hand is less desirable, since a user cannot create a demonstration while focused on driving, and other interactions such as speech would be more appropriate. Some of the technical approaches in my work like generalizing XPaths for element selection may be adapted if the target technology similarly exposes a tree-like representation of its UI, e.g., the view hierarchy in Android as used in Sugilite [88] and its follow-on work [90, 91, 89]. However, not all software or operating systems may be based on or expose a tree-like representation, so an implementation-agnostic approach for understanding and automating UIs based on computer vision may prove more feasible [110, 127, 121, 56, 66, 120, 119].

CHAPTER 7

Conclusion

In this thesis, I explored challenges of and novel approaches for creating web automation macros. First, I prototyped a web automation IDE and conducted two studies exploring how programmers write automation code – key findings include that identifying appropriate UI element selection logic is challenging, and that programmers benefit from having context about their target UI linked with their development environment. Next, I designed two programming by demonstration systems, ParamMacros and ScrapeViz, that enable users to create web automation macros without writing code. Both systems leverage user-provided demonstrations and structural patterns in the UI implementation to aid their generalization. ParamMacros specifically leverages one user-provided demonstration and user-provided parameters, and ScrapeViz leverages two user-provided examples of each kind of action to generalize. ScrapeViz also focuses on providing users tools for understanding a given macro – a high-level visual representation of the macro's sequence of actions and how they generalize across different website pages, and links between scraped data and their page source. Future work should continue exploring both developer tool and end-user programming approaches to support the broad range of people who may want to create custom web automation to improve their daily lives. In particular, future work should explore how to best support users when target websites change and macros need repair, and how to most effectively combine human and machine intelligence in authoring and editing macros.

BIBLIOGRAPHY

[1] Act-1: Transformer for actions. https://www.adept.ai/act. Accessed: 2022-11-28.

[2] Ad observatory by nyu tandon school of engineering: Explore facebook and instagram political ads. https://adobservatory.org/. Accessed: 2023-12-28.

[3] Adept - useful general intelligence. https://www.adept.ai/. Accessed: 2023-10-22.

[4] Amazon alexa voice ai. https://developer.amazon.com/en-US/alexa. Accessed: 2022-04-06.

[5] Americans use the internet to abandon children adopted from overseas. https://www.reuters.com/investigates/adoption/#article/part1. Accessed: 2023-12-28.

[6] App actions overview. https://developers.google.com/assistant/app/overview. Accessed: 2022-06-06.

[7] Attribute selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors. Accessed: 2021-05-05.

[8] Beautiful soup. https://www.crummy.com/software/BeautifulSoup/. Accessed: 2022-06-05.

[9] Class selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors. Accessed: 2021-05-05.

[10] Copy, paste, legislate. https://www.usatoday.com/pages/interactives/asbestos-sharia-law-model-bills-lobbyists-special-interests-influence-
Accessed: 2023-12-28.

[11] Cortana. https://www.microsoft.com/en-us/cortana. Accessed: 2022-04-06.

[12] The covid tracking project. https://covidtracking.com/. Accessed: 2023-12-28.

[13] Css selectors. https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors. Accessed: 2021-03-19.

[14] Cypress. https://www.cypress.io/. Accessed: 2021-03-19.

[15] Cypress studio. https://docs.cypress.io/guides/core-concepts/cypress-studio. Accessed: 2021-05-05.

[16] Document object model (dom). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/. Accessed: 2021-06-29.

[17] Dom property viewer. https://developer.mozilla.org/en-US/docs/Tools/DOM_Property_Viewer. Accessed: 2021-05-02.

[18] Electron. https://www.electronjs.org/. Accessed: 2020-09-18.

[19] Get started with viewing and changing the dom. https://developer.chrome.com/docs/devtools/dom/. Accessed: 2021-05-02.

[20] Google assistant. https://assistant.google.com/. Accessed: 2022-04-06.

[21] Id selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/ID_selectors. Accessed: 2021-05-05.

[22] imacros. https://imacros.net/. Accessed: 2020-06-08.

[23] Internet archive wayback machine. https://archive.org/web/. Accessed: 2021-05-05.

[24] The lost homes of detroit: Hundreds of millions of dollars in tax debt that forced detroiters out of their homes never should have been charged in the first place. https://revealnews.org/podcast/the-lost-homes-of-detroit/. Accessed: 2023-12-28.

[25] Monacoeditor. https://microsoft.github.io/monaco-editor/. Accessed: 2021-05-02.

[26] Multion - your personal ai agent. https://www.multion.ai/. Accessed: 2023-10-22.

[27] The new york times developer network. https://developer.nytimes.com/. Accessed: 2023-12-29.

[28] The only newsroom dedicated to covering gun violence. https://www.thetrace.org/2020/01/armslist-unlicensed-gun-sales-engaged-in-the-business/. Accessed: 2023-12-28.

[29] Playwright. https://playwright.dev/. Accessed: 2022-11-30.

[30] Public domain data collectors for the work of congress, including legislation, amendments, and votes. https://github.com/unitedstates/congress. Accessed: 2023-12-28.

[31] Puppeteer. https://pptr.dev/. Accessed: 2020-09-18.

[32] Puppeteer waitfornavigation. https://pptr.dev/#?product=Puppeteer&version=v8.0.0&show=api-pagewaitfornavigationoptions/. Accessed: 2021-05-02.

[33] React: The library for web and native user interfaces. https://react.dev/. Accessed: 2024-01-07.

[34] rrweb-snapshot. https://github.com/rrweb-io/rrweb-snapshot. Accessed: 2020-09-18.

[35] Selenium. https://www.selenium.dev/. Accessed: 2020-09-11.

[36] Selenium ide. https://www.selenium.dev/selenium-ide/. Accessed: 2020-06-08.

[37] Shortcuts user guide. https://support.apple.com/guide/shortcuts/welcome/ios. Accessed: 2022-06-06.

[38] Siri. https://www.apple.com/siri/. Accessed: 2022-04-06.

[39] To protect and slur: Inside hate groups on facebook, police officers trade racist memes, conspiracy theories and islamophobia. https://revealnews.org/topic/to-protect-and-slur/. Accessed: 2023-12-28.

[40] Web embeds. https://www.electronjs.org/docs/latest/tutorial/web-embeds. Accessed: 2023-10-21.

[41] webcontents, event: 'will-navigate'. https://www.electronjs.org/docs/latest/api/web-contents#event-will-navigate. Accessed: 2024-01-07.

[42] Webflow: Create a custom website — no-code website builder. https://webflow.com/. Accessed: 2023-10-22.

[43] What is an api? https://www.ibm.com/topics/api. Accessed: 2023-12-27.

[44] What is javascript? https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript. Accessed: 2022-11-28.

[45] Why web scraping is vital to democracy. https://themarkup.org/news/2020/12/03/why-web-scraping-is-vital-to-democracy. Accessed: 2023-12-28.

[46] Xpath. https://developer.mozilla.org/en-US/docs/Web/XPath/. Accessed: 2021-03-20.

[47] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28(5):793–819, 2019.

[48] Frank Andrade. Web scraping with chatgpt. https://www.linkedin.com/posts/thepycoach_web-scraping-with-chatgpt-code-interpreter-activity-7088193
Accessed: 2023-10-23.

[49] Deniz Arsan, Ali Zaidi, Aravind Sagar, and Ranjitha Kumar. App-based task shortcuts for virtual assistants. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 1089–1099, 2021.

[50] Jeffrey P Bigham and Richard E Ladner. Accessmonkey: a collaborative scripting framework for web users and developers. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 25–34, 2007.

[51] Jeffrey P Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: enabling blind users to blaze trails through the web. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 177–186, 2009.

[52] Jeffrey P Bigham, Irene Lin, and Saiph Savage. The effects of" not knowing what you don't know" on web accessibility for blind web users. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*, pages 101–109, 2017.

[53] Alan F Blackwell. Swyn: A visual representation for regular expressions. In *Your wish is my command*, pages 245–XIII. Elsevier, 2001.

[54] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1535–1544, 2010.

[55] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 963–975, 2018.

[56] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Towards complete icon labeling in mobile applications. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2022.

[57] Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. Miwa: Mixed-initiative web automation for better user control and confidence. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023.

[58] Zhutian Chen and Haijun Xia. Crossdata: Leveraging text-data connections for authoring data documents. In *CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2022.

[59] Pei-Yu Chi, Sen-Po Hu, and Yang Li. Doppio: Tracking ui flows and code changes for app development. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[60] Pei-Yu Chi, Yang Li, and Björn Hartmann. Enhancing cross-device interaction scripting with interactive illustrations. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5482–5493, 2016.

[61] Mark Craven, Andrew McCallum, Dan PiPasquo, Tom Mitchell, and Dayne Freitag. Learning to extract symbolic knowledge from the world wide web. Technical report, Carnegie Mellon University, 1998.

[62] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[63] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854, 2017.

[64] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. Webrobot: Web robotic process automation using interactive programming-by-demonstration. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

[65] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12, 2020.

[66] Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. Understanding screen relationships from screenshots of smartphone applications. In *27th International Conference on Intelligent User Interfaces*, pages 447–458, 2022.

[67] Michael H Fischer, Giovanni Campagna, Euirim Choi, and Monica S Lam. Diy assistant: a multi-modal end-user programmable virtual assistant. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 312–327, 2021.

[68] Martin R Frank, Piyawadee" Noi" Sukaviriya, and James D Foley. Inference bear: designing interactive interfaces through before and after snapshots. In *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*, pages 167–175, 1995.

[69] Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023.

[70] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G Karahalios. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th annual acm symposium on user interface software & technology*, pages 489–500, 2015.

[71] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[72] Brian Hempel and Ravi Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 379–390, 2016.

[73] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-sketch: Output-directed programming for svg. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 281–292, 2019.

[74] Chris Hess and Sarah E Chasins. Informing housing policy through web automation: Lessons for designing programming tools for domain experts. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1–9, 2022.

[75] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. Extending manual drawing practices with artist-centric programming tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[76] Jennifer Jacobs, Sumit Gogia, Radomír Měch, and Joel R Brandt. Supporting expressive procedural art creation through direct manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 6330–6341, 2017.

[77] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. Kitty: sketching dynamic and interactive illustrations. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 395–405, 2014.

[78] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. Draco: bringing life to illustrations with kinetic textures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 351–360, 2014.

[79] Rubaiat Habib Kazi, Tovi Grossman, Hyunmin Cheong, Ali Hashemi, and George W Fitzmaurice. Dreamsketch: Early stage 3d design explorations with sketching and generative design. In *UIST*, volume 14, pages 401–414, 2017.

[80] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid moves between code and graphical work in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 140–151, 2020.

[81] Rebecca Krosnick, Sang Won Lee, Walter S Laseck, and Steve Oney. Expresso: Building responsive interfaces with keyframes. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 39–47. IEEE, 2018.

[82] Rebecca Krosnick and Steve Oney. Parammacros: Creating ui automation leveraging end-user natural language parameterization. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10. IEEE, 2022.

[83] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: insights from the practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1090–1101. IEEE, 2018.

[84] Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–65, 2009.

[85] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534. Citeseer, 2000.

[86] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2020.

[87] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728, 2008.

[88] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. Sugilite: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 6038–6049, 2017.

[89] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M Mitchell, and Brad A Myers. Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 1094–1107, 2020.

[90] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. Appinite: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–114. IEEE, 2018.

[91] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*, pages 577–589, 2019.

[92] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences. In *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.

[93] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[94] Geoffrey Litt and Daniel Jackson. Wildcard: spreadsheet-driven customization of web applications. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 126–135, 2020.

[95] Greg Little, Tessa A Lau, Allen Cypher, James Lin, Eben M Haber, and Eser Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, 2007.

[96] Sahisnu Mazumder and Oriana Riva. Flin: A flexible natural language interface for web navigation. In *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, June 2021.

[97] Richard G McDaniel and Brad A Myers. Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 442–449, 1999.

[98] Jan Meskens, Kris Luyten, and Karin Coninx. D-macs: building multi-device user interfaces by demonstrating, sharing and replaying design actions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 129–138, 2010.

[99] Robert C Miller and Brad A Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.

[100] Francesmary Modugno and Brad A Myers. A state-based visual language for a demonstrational visual shell. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 304–311. IEEE, 1994.

[101] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. recode: A lightweight find-and-replace interaction in the ide for transforming code by example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 258–269, 2021.

[102] Donald A Norman. *The psychology of everyday things.* Basic books, 1988.

[103] Steve Oney, Alan Lundgard, Rebecca Krosnick, Michael Nebeling, and Walter S Lasecki. Arboretum and arbility: Improving web accessibility through a shared browsing architecture. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 937–949, 2018.

[104] Lihang Pan, Chun Yu, JiaHui Li, Tian Huang, Xiaojun Bi, and Yuanchun Shi. Automatically generating and improving voice command interface from operation sequences on smartphones. In *CHI Conference on Human Factors in Computing Systems*, pages 1–21, 2022.

[105] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. Gui-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering*, 48(3):910–929, 2020.

[106] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. Semanticon: Specifying content-based semantic conditions for web automation programs. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16, 2022.

[107] Kevin Pu, Jim Yang, Angel Yuan, Minyi Ma, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. Dilogics: Creating web automation programs with diverse logics. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023.

[108] Oriana Riva and Jason Kace. Etna: Harvesting action graphs from websites. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 312–331, 2021.

[109] Christopher Scaffidi, Allen Cypher, Sebastian Elbaum, Andhy Koesnandar, and Brad Myers. Scenario-based requirements for web macro tools. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 197–204. IEEE, 2007.

[110] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. Vasta: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th international conference on intelligent user interfaces*, pages 22–32, 2020.

[111] Vidya Setlur, Sarah E Battersby, Melanie Tory, Rich Gossweiler, and Angel X Chang. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th annual symposium on user interface software and technology*, pages 365–377, 2016.

[112] Vidya Setlur, Enamul Hoque, Dae Hyun Kim, and Angel X Chang. Sneak pique: Exploring autocompletion as a data discovery scaffold for supporting visual analysis. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 966–978, 2020.

[113] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 503–514, 2018.

[114] Khai N Truong, Gillian R Hayes, and Gregory D Abowd. Storyboarding: an empirical determination of best practices and effective guidelines. In *Proceedings of the 6th conference on Designing Interactive systems*, pages 12–21, 2006.

[115] Bret Victor. Inventing on principle. `https://www.youtube.com/watch?v=PUv66718DII`. Accessed: 2022-12-01.

[116] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[117] Nora S Willett, Rubaiat Habib Kazi, Michael Chen, George Fitzmaurice, Adam Finkelstein, and Tovi Grossman. A mixed-initiative interface for animating static pictures. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 649–661, 2018.

[118] 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 systems*, pages 1435–1444, 2007.

[119] Jason Wu, Rebecca Krosnick, Eldon Schoop, Amanda Swearngin, Jeffrey P Bigham, and Jeffrey Nichols. Never-ending learning of user interfaces. 2023.

[120] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. Webui: A dataset for enhancing visual ui understanding with web semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2023.

[121] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483, 2021.

[122] Haijun Xia, Nathalie Henry Riche, Fanny Chevalier, Bruno De Araujo, and Daniel Wigdor. Dataink: Direct and creative data-oriented drawing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2018.

[123] Jun Xing, Rubaiat Habib Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. Energy-brushes: Interactive tools for illustrating stylized elemental dynamics. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 755–766, 2016.

[124] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. Guider: Gui structure and vision co-guided test script repair for android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 191–203, 2021.

[125] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009.

[126] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, 2020.

[127] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[128] Fengbin Zhu, Wenqiang Lei, Chao Wang, Jianming Zheng, Soujanya Poria, and Tat-Seng Chua. Retrieving and reading: A comprehensive survey on open-domain question answering. *arXiv preprint arXiv:2101.00774*, 2021.

[129] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. Lyra 2: Designing interactive visualizations by demonstration. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):304–314, 2020.