MIWA: Mixed-Initiative Web Automation for Better User Control and Confidence

Weihao Chen
chen4129@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Xiaoyu Liu
dawnsqrl@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Jiacheng Zhang
jiache@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

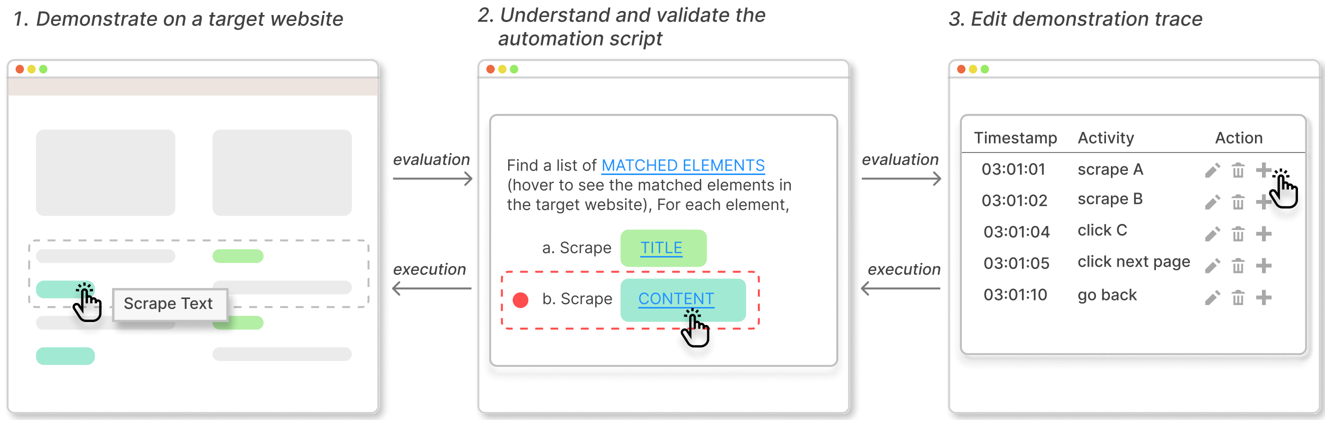
Ian Iong Lam
iinicole@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Zhicheng Huang
skyhuang@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Rui Dong
ruidong@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Xinyu Wang
xwangsd@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Tianyi Zhang
tianyi@purdue.edu
Purdue University
West Lafayette, Indiana, USA


1. Demonstrate on a target website
Scrape Text
evaluation
execution
2. Understand and validate the automation script
a. Scrape
TITLE
b. Scrape
CONTENT
evaluation
execution
3. Edit demonstration trace

Find a list of MATCHED ELEMENTS (hover to see the matched elements in the target website), For each element,

Timestamp	Activity	Action
03:01:01	scrape A	✎ 🗑 +
03:01:02	scrape B	✎ 🗑 +
03:01:04	click C	✎ 🗑 +
03:01:05	click next page	✎ 🗑 +
03:01:10	go back	✎ 🗑 +

Figure 1: Given the user's demonstration on a target website, MIWA synthesizes an automation script as well as a step-by-step explanation of the script's behavior. Users can easily understand the synthesized script by reading the natural language (NL) description. When users hover over an entity (e.g., a column name) in the NL description, the corresponding elements on the target website are highlighted to help users see the visual correspondence between the NL description and the web elements. Users can further validate the correctness of each step in the automation script via a step-through debugging feature. If an error is identified, users can undo, redo, or edit previous demonstrations to refine the script.

ABSTRACT

In the era of Big Data, web automation is frequently used by data scientists, domain experts, and programmers to complete time-consuming data collection tasks. However, developing web automation scripts requires familiarity with a programming language and HTML, which remains a key learning barrier for non-expert users. We provide MIWA, a mixed-initiative web automation system that enables users to create web automation scripts by demonstrating what content they want from the targeted websites. Compared to existing web automation tools, MIWA helps users better understand a generated script and build trust in it by (1) providing a step-by-step explanation of the script's behavior with visual correspondence to the target website, (2) supporting greater autonomy and control over web automation via step-through debugging and fine-grained demonstration refinement, and (3) automatically detecting potential corner cases that are handled improperly by the generated script. We conducted a within-subjects user study with 24 participants and compared MIWA with Rousillon, a state-of-the-art web automation tool. Results showed that, compared to Rousillon, MIWA reduced

UIST '23, October 29–November 01, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0132-0/23/10.
https://doi.org/10.1145/3586183.3606720

the task completion time by half while helping participants gain more confidence in the generated script.

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; **Web-based interaction**; **Interaction design**; **Systems and tools for interaction design**.

KEYWORDS

Programming by Demonstration, Web Automation, Data Science

ACM Reference Format:
Weihao Chen, Xiaoyu Liu, Jiacheng Zhang, Ian Iong Lam, Zhicheng Huang, Rui Dong, Xinyu Wang, and Tianyi Zhang. 2023. MIWA: Mixed-Initiative Web Automation for Better User Control and Confidence. In *The 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23), October 29–November 01, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3586183.3606720

1 INTRODUCTION

In the era of Big Data, data scientists, domain experts, and programmers collect data to construct data-driven applications [20, 24, 63]. For example, social scientists rely heavily on web data (e.g., tweets, forum posts) to gain insights into social behavior, opinion development, and cultural preferences [12, 43]. To gather such web data, they need to create a web automation script [16], since manual data collection is time-consuming and prone to human errors [13]. However, implementing web automation scripts requires users to understand web browser events and reverse-engineer target websites [16]. Additionally, they should be proficient in web programming languages (e.g., HTML, DOM, and JavaScript), as well as web automation libraries such as Selenium [9], Scrapy [8], Beautiful Soup [1], and MechanicalSoup [6].

Programming by Demonstration (PBD) is a promising solution for web automation without requiring web programming skills [32]. Though many PBD systems [4, 10, 16, 17, 35, 36, 40, 41] have been proposed for web automation, existing systems provide limited support for helping users understand, validate, and build confidence on the generated scripts. As shown by Krosnick and Oney [29], users desire to receive more live feedback when debugging their scripts. Furthermore, in case of errors, users have no choice but to restart from scratch, since current PBD systems only support one-shot synthesis. There is a lack of effective mechanisms for discovering, diagnosing, and repairing errors in the generated scripts.

To address these limitations, we present MIWA, a mixed-initiative system that facilitates the comprehension, validation, and refinement of web automation scripts with increased user control and autonomy. To help users understand and validate a generated script, we develop a grammar-based method that generates a step-by-step description of the script's behavior in natural language (NL). To improve the readability of the NL description, MIWA further augments the description with visual correspondence between entities mentioned in the description and UI elements on the target website. Users can validate the script's behavior by inspecting the intermediate result of each web automation step via a step-through debugging feature. If the NL description is not aligned with the user's intent, they can further demonstrate more actions or modify previous actions in the demonstration trace without the need to start over. To speed up the synthesis efficiency, MIWA leverages a novel incremental synthesis method to monitor user actions and continuously refine the script from previous checkpoints instead of re-synthesizing from scratch. Finally, when MIWA encounters non-existing or incorrectly formatted data, it immediately prompts users with an alert and solicits fixes to avoid processing more data incorrectly.

We conducted a within-subjects user study with 24 participants to evaluate the usability and efficiency of MIWA. Participants using MIWA finished the assigned tasks in only 3 minutes and 58 seconds on average, reducing the task completion time by 55% compared with using Rousillon [16]. In the post-task survey, participants felt more confident about the generated scripts when using MIWA—(5.58 vs. 6.50, unpaired t-test: p=.00189) on average on a 7-point scale. These results imply that MIWA can indeed improve users' productivity and confidence in web automation. Besides, we conducted a qualitative experiment on 29 additional tasks to demonstrate the effectiveness and generalizability of MIWA. The average task completion time is 2 minutes and 50 seconds, with a success rate of 74%. This provides quantitative evidence about MIWA's effectiveness on a variety of web automation tasks.

Overall, this work makes the following contributions:

- MIWA, a mixed-initiative interaction system that enables users to build web automation scripts with better user control and confidence.
- A within-subjects user study demonstrating the usability and efficacy of MIWA in comparison to a state-of-the-art tool called Rousillon.
- A quantitative experiment on 29 web automation tasks demonstrating the effectiveness and generalizability of MIWA.

2 RELATED WORK

2.1 Web Automation

Web automation is a widely adopted technique that uses software scripts to automate tedious and error-prone web-related tasks, such as filling in web forms and collecting data from websites. For instance, social scientists may need to write web automation scripts in order to collect data from various sources, such as social media networks like Reddit and Twitter [65]. Investors may use automation to collect stock data in order to monitor and analyze market trends [31]. Web automation is also used to perform other domain-specific tasks, such as gathering a large amount of training data to curate machine learning models and automatically testing UI components [28, 62].

However, implementing web automation scripts is a difficult and complex endeavor. It requires expertise in HTML/DOM, CSS, programming, and computational thinking. Krosnick and Oney [29] identify a list of key challenges for developing web automation scripts. They find that even for seasoned programmers, it takes a considerable amount of time to understand the web structure, its contents, and the behavior of UI elements before they can start writing code to interact with the websites.

Several techniques and tools have been developed to reduce the technical barrier for authoring web automation scripts. For example, Selenium [9], Puppeteer [7], and Cypress [3] are three commercial tools that use record-and-replay to reproduce user actions. However, these techniques are not capable of generalizing beyond what has been demonstrated. In other words, the scripts generated by these techniques only replay the user actions but cannot be applied to similar elements on a target website. Thus, they still require a considerable amount of manual effort to edit the generated scripts to generalize beyond user demonstrations. CoScripter [36] (formerly Koala [41]) records user demonstrations as pseudo-natural language scripts and leverages sloppy programming to interpret the scripts. It only supports limited generalization, e.g., replacing a literal value with a variable if it appears in a database. More recent tools [14–17, 21, 29, 52, 53] use advanced program synthesis algorithms to generate web automation scripts with variables and loops, which can generalize beyond user demonstrations and apply to similar elements on the same website across multiple pages. We describe these techniques in the next section.

2.2 Programming-by-Demonstration (PBD)

Programming-by-Demonstration (PBD) is a class of program synthesis techniques that automatically generate programs from user demonstrations. PBD techniques have shown success in lowering the technical barrier and making programming accessible to non-programmers [16, 19, 23, 33, 37–40, 44, 46, 47, 51]. For example, one of the earliest PBD techniques, Pedriot [50], allows users to create UI components (e.g., menus and scroll bars) by demonstrating how they would create them manually.

The most related to us are PBD systems for web automation [14–17, 21, 29, 52, 53]. Given a sequence of user actions on a website, these techniques automatically generate scripts that repeat the same actions and can also apply them to similar elements in the website. Some PBD tools [22, 35, 64] can only generate web automation scripts with one-level loops but not nested loops, which limits their capability to automate complex actions. To address this limitation, Chasins et al. proposed Rousillon [16]. Furthermore, to improve the readability of generated scripts, Rousillon adopts the visual syntax of Scratch [55] to render the generated scripts. While the Scratch-style visualization makes it easier to read a script, it still requires users to know programming basics, such as loops and variables, to understand and edit the script. Furthermore, Rousillon assigns some mechanically generated names such as `row_1` and `list_2` in a script, which are hard to interpret. Dong et al. [17] recently presented a new PBB algorithm called WebRobot that leverages speculative rewriting to generate web automation scripts with nested loops. WebRobot allows users to execute a generated script and continuously provide more demonstrations to refine it. However, it does not render the generated script to users or provide a description of the script's behavior. Therefore, it is hard for users to validate the script other than running the script on the entire website and manually checking the scraped result.

Compared to previous work, our main focus is to design interaction mechanisms that promote user trust and control over web automation scripts. Specifically, our system provides a step-by-step description of a generated script and allows users to execute each step one by one to validate its behavior. Our user study results show that these mechanisms significantly reduce task completion time and improve user confidence compared to Rousillon [16].

2.3 Interactive Support in Existing PBD Systems

Existing PBD systems adopt various interaction mechanisms to improve the usability of PBD. For instance, Krosnick and Oney recently proposed a tool to allow users to specify which parts of a natural language query can be generalized, in addition to providing demonstration traces [30]. This helps PBD systems better reduce ambiguity in user demonstrations. Such multimodal specifications have also been adopted by other PBD systems [23, 37–39, 49]. PUMICE [39], APPINITE [38], and SUGILITE [37] allow users to resolve ambiguities and vagueness in a task description via both conversations and demonstrations. Topaz [49] allows users to specify which parts of a demonstration can be generalized by pointing to the objects that should be generalized. Vegemite [40] uses both demonstrations and direct manipulations to populate tables with information collected from different websites. HILC [23] asks users to answer some follow-up questions after a demonstration to help clarify the confusing parts in the demonstration.

The interaction mechanisms most related to our work are tools designed to promote user trust and confidence in PBD [16, 36, 42, 47]. As Tessa Lau wrote in her reflection on why PBD fails [34], PBD systems should "*encourage trust by presenting a user-friendly model.*" Visual programming promotes user understanding and trust by providing live feedback to users [61]. For instance, VIVA [61] renders an image processing script as an executable flowchart and provides live feedback at four different levels. Similarly, Rousillon [16] renders a web automation script in a block-based visual language [55]. Pursuit [47] renders the generated program in a comic strip style. In this work, we choose to use natural language rather than a graphical representation to help users understand a program, since we find natural language more suitable and straightforward for describing actions on a webpage, without requiring graphic literacy.

Similar to our work, both CoScripter [36] and FlashProg [42] translate a generated program into pseudo-natural language. However, CoScripter [36] only performs a verbatim translation of individual user actions in a demonstration trace and does not handle complex program structures like loops. FlashProg [42] uses templates and idiom paraphrasing rules to translate synthesized regular expressions. Compared with regular expressions, web automation scripts are more complex, with variables and loops to explain. MIWA adopts a grammar-based method to systematically translate different parts of a program based on its abstract syntax tree, instead of only relying on heuristics. Additionally, MIWA not only generates a plain text description for a program but also renders the visual correspondence between elements mentioned in a program and the elements in a target website.

The design of fine-grained control features in MIWA is highly motivated by the idea of direct manipulation [57, 58]. Direct manipulation has been widely adopted in different domains, such as content creation [11, 25, 26], visual programming [48, 55], and data visualization [27, 60]. Rousillon [16] also supports direct manipulation by rendering the generated scripts in a block-based visual language and allowing users to edit them directly. However, we found that such a visual language still requires some basic understanding of programming, such as understanding variables, loops, and type compatibility. MIWA addresses this limitation by allowing users to edit demonstration traces rather than generated scripts. Once a demonstration is refined, MIWA will automatically update the underlying script accordingly.

D-Macs [45] is another work highly related to MIWA. D-Macs supports record-and-replay for multi-device UI design with similar features, such as visualizing the demonstration trace and handling potential errors. The main difference between MIWA and D-Macs is that MIWA can produce a natural language explanation of the synthesized script in a step-by-step manner. This feature helps users comprehend and debug the script. Additionally, the error handling mechanism in MIWA focuses on data anomalies, while D-Macs focuses on checking whether an action is applicable to a new device, as its goal is multi-device UI design.

3 DESIGN GOALS AND SYSTEM OVERVIEW

We reviewed previous studies [16, 21, 29, 34] that have conducted user studies of web automation tools or discussed the usability challenges of web automation. Based on the common issues and challenges, we summarized the design goals for MIWA and elaborated on the rationale for each goal below.

D1. Help users understand the generated script. Several studies have shown that users wish to have a better way of understanding the script generated by a web automation tool [16, 21, 34]. For example, Chasins et al. [16] showed that participants appreciated having a more readable language in Rousillon compared to a low-level script language in Selenium. In another article [34] that reflects the lessons learned from deploying PBD systems, including CoScripter [36], Tessa Lau reported that many PBD systems were not adopted since users cannot grasp the arcane syntax of generated programs. This difficulty refrains users from fully trusting generated programs.

These findings motivate us to design the grammar-based translation method to explain a generated script in natural language (Section 5.2). Compared to other kinds of program representations, such as a visual language [55], natural language can be more easily understood by humans with little requirement for graph literacy.

D2. Provide live feedback. Krosnick and Oney [29] found that users of web automation tools wish to have live feedback to understand the web automation process and discover problems. Specifically, users want to see which HTML elements are matched by a CSS selector (i.e., XPath) in a script. They also want to see the intermediate outcome of each step of the execution to understand whether the correct elements are selected, whether the expected behavior occurs, and whether there are any errors.

This motivates us to design two interactive features in MIWA. First, MIWA allows users to hover over an entity in a NL explanation to see the visual correspondence between the entity and the HTML elements matched by the CSS selector of the entity (Section 5.2). Second, MIWA supports step-through debugging, a well-established feature in programming IDEs, to help users pause program execution and inspect the intermediate result of each statement in a script (Section 5.3).

D3. Help users detect potential errors. Chasins et al. [16] pointed out that many Selenium users had concerns about the robustness of web automation scripts. Lau [34] also showed that CoScripter [36] was likely to crash in the middle of execution, as it relied on heuristics rather than formal syntax for parsing. She found that users became confused and had no choice but to restart the task. Thus, Lau suggested that PBD systems should help users detect potential errors and fail gracefully. Krosnick and Oney [29] summarized several root causes for web automation errors, such as HTML/CSS element inconsistencies across pages, learning a CSS selector that is too general or too specific, etc.

Inspired by these findings, we designed an anomaly detection feature that proactively detects common data anomalies in web automation (Section 5.4). Furthermore, the step-through debugging feature (Section 5.3) can help users accurately locate which step of the execution leads to an error and make targeted corrections.

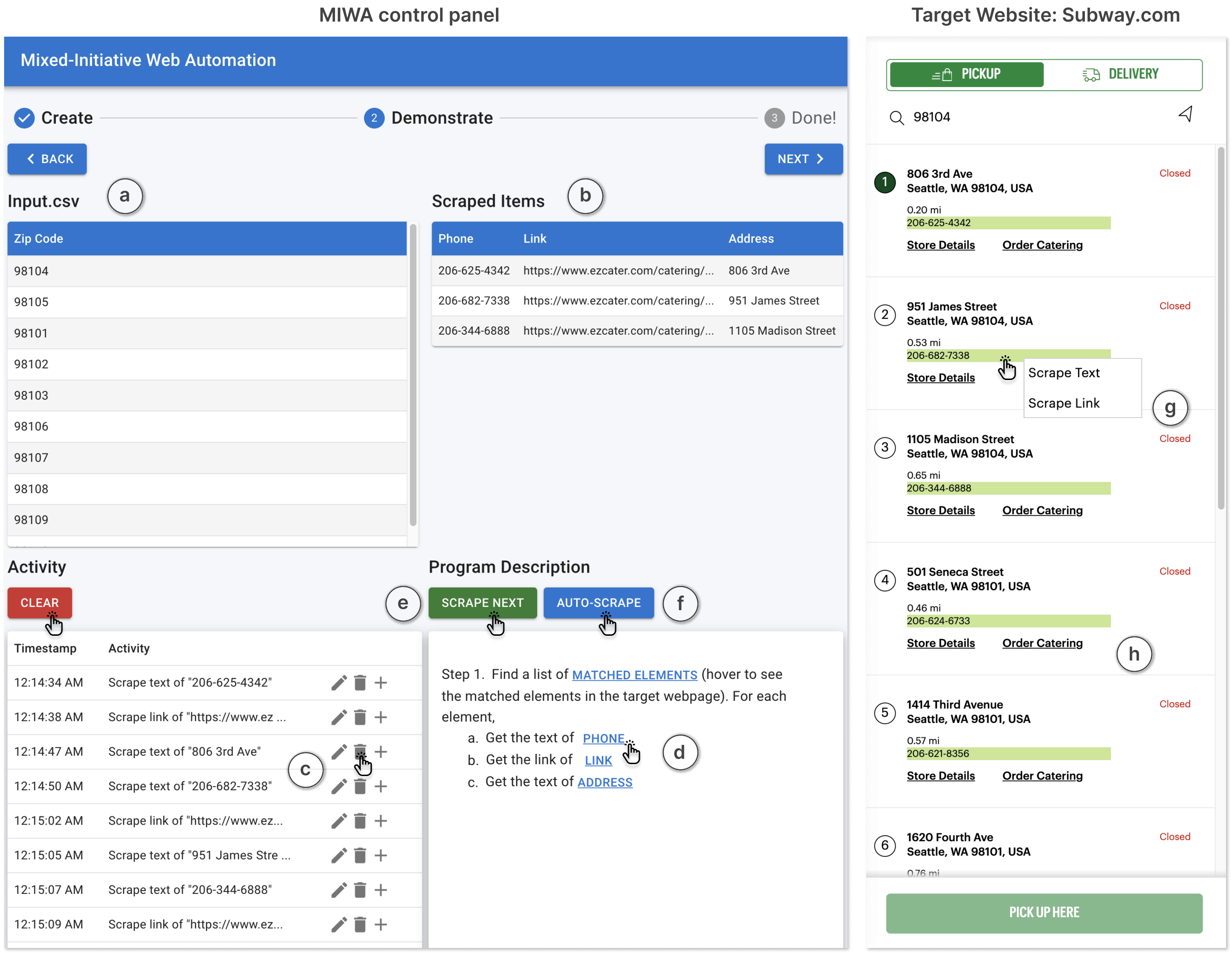
D4. Make it easier to make corrections. Several studies have found that users wish to have more convenient ways to fix errors in the scripts generated by a PBD system [16, 21, 34]. In PBD, the default error-fixing mechanism is to provide new demonstration traces. However, this is not efficient, especially for errors that require a small change, since users have to demonstrate from the very beginning [34]. While some tools, such as Selenium, allow users to directly edit the generated script to fix an error, they also introduce a steep learning curve, since users need to be familiar with the script language to make the correction.

The user study by Chasins et al. [16] confirms that participants using Selenium complained about the challenges of interacting with a low-level script language. Rousillon [16] addresses this challenge by rendering scripts in a block-based visual language. However, we found that this visual language still required a basic understanding of program constructs, such as variables and loops, and the visualization can be overwhelming for complex scripts. Thus, we chose to allow users to edit individual actions in a demonstration trace rather than the underlying script to fix an error (Section 5.3). MIWA will automatically update the script based on the refined trace.

4 USAGE SCENARIO

Suppose Alice is a social scientist who wants to gather data from the Subway website in order to analyze the distribution of local Subway stores and its relation to the poverty of neighborhoods. Without any tool support, Alice needs to go to the Subway website, enter a zip code, and manually copy and paste the information of each nearby Subway store in the search result. She has to repeat these steps for all neighborhoods she cares about. This is time-consuming and error-prone. Since Alice is not familiar with web programming, she does not know how to write a script to automate this process. Therefore, Alice decides to use MIWA to automatically generate the web automation script she needs.

Alice's task is to collect the phone numbers, hyperlinks, and addresses of local Subway stores in different neighborhoods. She has curated a spreadsheet with a list of zip codes for the neighborhoods of interest. Alice uploads this spreadsheet to MIWA, which is rendered in an input table in Figure 2ⓐ. Then, Alice drags the first zip code and drops it into the search box of the target website. After the search results are returned, she right-clicks on the phone number of



Input.csv

Zip Code
98104
98105
98101
98102
98103
98106
98107
98108
98109

Scraped Items

Phone	Link	Address
206-625-4342	https://www.ezcater.com/catering/...	806 3rd Ave
206-682-7338	https://www.ezcater.com/catering/...	951 James Street
206-344-6888	https://www.ezcater.com/catering/...	1105 Madison Street

Activity

Timestamp	Activity
12:14:34 AM	Scrape text of "206-625-4342"
12:14:38 AM	Scrape link of "https://www.ez ...
12:14:47 AM	Scrape text of "806 3rd Ave"
12:14:50 AM	Scrape text of "206-682-7338"
12:15:02 AM	Scrape link of "https://www.ez...
12:15:05 AM	Scrape text of "951 James Stre ...
12:15:07 AM	Scrape text of "206-344-6888"
12:15:09 AM	Scrape link of "https://www.ez...

Program Description

Step 1. Find a list of MATCHED ELEMENTS (hover to see the matched elements in the target webpage). For each element,

a. Get the text of PHONE
b. Get the link of LINK
c. Get the text of ADDRESS

98104

1 806 3rd Ave
Seattle, WA 98104, USA
0.20 mi
206-625-4342
Store Details Order Catering
Closed

2 951 James Street
Seattle, WA 98104, USA
0.53 mi
206-682-7338
Store Details
Closed

3 1105 Madison Street
Seattle, WA 98104, USA
0.65 mi
206-344-6888
Store Details Order Catering
Closed

4 501 Seneca Street
Seattle, WA 98101, USA
0.46 mi
206-624-6733
Store Details Order Catering
Closed

5 1414 Third Avenue
Seattle, WA 98101, USA
0.57 mi
206-621-8356
Store Details Order Catering
Closed

6 1620 Fourth Ave
Seattle, WA 98101, USA
0.76 mi
Closed

Figure 2: The user interface of MIWA. In addition to scraping data (ⓑ and ⓖ), users can fill out a text field (e.g., a search bar) on a target website by dragging values from a csv file uploaded to MIWA (ⓐ). MIWA provides a step-by-step explanation of the synthesized script with visual correspondence to the UI elements in the target website (ⓓ and ⓗ). Users can validate the script behavior and inspect intermediate automation results through a step-by-step debugging feature (ⓔ). If users spot any error, they can refine the demonstration trace by editing previous actions or demonstrating new actions (ⓒ). In the end, users can scrape the data all at once by clicking the AUTO-SCRAPE button (ⓕ).

the first Subway store and selects the "Scrape Text" option (Figure 2ⓖ). Now she can see that the phone number has been scraped and put into the output table (Figure 2ⓑ). The activity trace in Figure 2ⓒ is also updated with an action description—"*Scrape text of 206-625-4342*." Alice confirms that her action has been recorded correctly. She then proceeds to scrape the hyperlink and address of the store. During the demonstration, she mistakenly scrapes the distance information (0.2 mi) as part of the store address. MIWA soon detects a wrong format and displays a potential error alert (Figure 6ⓓ) to her (**D3**). To fix this mistake, Alice clicks the edit button (Figure 2ⓒ) in the activity trace and redoes this action (**D4**).

After Alice scrapes the data for two Subway stores, she notices that the synthesizer has generated a web automation script. She quickly understands the natural language description of this script (**D1**), as shown in Figure 2ⓓ).

When Alice hovers over the column name "PHONE" (Figure 2ⓓ), all the phone numbers on the target website are then highlighted (Figure 2ⓗ). This visual correspondence helps Alice confirm that MIWA's understanding of PHONE is aligned with her expectation (**D2**). To double check whether this script behaves correctly (**D2**), she clicks the SCRAPE NEXT button several times and confirms that the next few elements are scraped correctly (Figure 2ⓔ). Alice is satisfied with the script, so she clicks the AUTO-SCRAPE button (Figure 2ⓕ).

MIWA scrapes all remaining data on the first page and then prompts Alice that the scraping is done (Figure 6ⓐ). Alice realizes that this script only scrapes data on the first page and does not know how to move to the next page (**D3**). So Alice continues to demonstrate by clicking the Next Page button and scraping the data for the first Subway store on the second page. After demonstrating, MIWA generates a new script with the following description:

> Step 1. Find a list of MATCHED ELEMENTS (hover to see the matched elements on the target page) For each element,
>
> a. Get the text of PHONE
> b. Get the text of LINK
> c. Get the text of ADDRESS
>
> Step 2. Click NEXT PAGE
> Step 3. Repeat the previous step(s)

Alice confirms that this script can automatically click the Next Page button and scrap data for all pages. Alice clicks the AUTO-SCRAPE button again. The script successfully iterates over all pages and scrapes all the data.

Now Alice wants to scrape the data from a different neighborhood. She drags the second zip code from the input table in Figure 2ⓐ and repeats the same demonstration again. This time, MIWA generates a new program with the following description:

> Step 1. Find a list of Zip Code from the input file. For each Zip Code,
>
> a. Enter Zip Code to SEARCH BOX
> b. Click SEARCH
> c. Find a list of MATCHED ELEMENTS (hover to see the matched elements on the target page). For each element,
>
> c1. Get the text of PHONE
> c2. Get the text of LINK
> c3. Get the text of ADDRESS
>
> d. Click NEXT PAGE
> e. Repeat the previous step(s)

Alice clicks the AUTO-SCRAPE button again. MIWA then scrapes all remaining data for the second neighborhood and iterate over the remaining neighborhoods listed in the input table in Figure 2ⓐ. In this way, Alice gathers all the data she needs for the analysis, without writing a single line of code.

5 SYSTEM IMPLEMENTATION

Figure 3 shows the system architecture of MIWA. MIWA includes three components: (1) a React UI that renders the input data, the scraped data, the demonstration trace, and the NL explanation; (2) a browser extension that records user demonstrations and highlights web elements mentioned in the NL explanation; (3) a back-end server that runs the PBD algorithm and detects potential anomalies. This section describes the key features of these components.

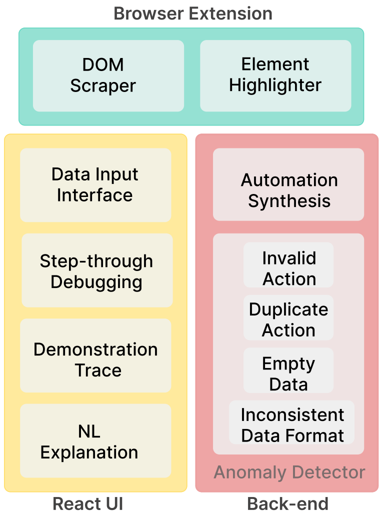


Figure 3: The system architecture of MIWA

5.1 Synthesizing Web Automation Scripts

MIWA leverages a state-of-the-art synthesis algorithm [17] to automatically generate web automation scripts from user-demonstrated actions. We briefly describe the synthesizer here and refer interested readers to [17] for details. The synthesizer takes two inputs: (1) user demonstration in the form of a trace of actions and (2) input data in the form of a JSON or CSV file. The input data is optional, but it is typically needed for form-filling tasks (e.g., entering search keywords on an online shopping website). The synthesizer returns a web automation script that repeats the steps in the user demonstration and continues to perform more actions on similar web elements.

$$
\begin{array}{llll}
\textit{Program} & P & ::= & S; \cdots; S \\
\textit{Statement} & S & ::= & \textit{Click}(n) \mid \textit{ScrapeText}(n) \mid \textit{ScrapeLink}(n) \\
 & & \mid & \textit{GoBack} \mid \textit{SendData}(n, v) \mid \textit{SendKeys}(n, s) \\
 & & \mid & \texttt{ForEach}\ y\ \texttt{in}\ N\ \texttt{do}\ P \\
 & & \mid & \texttt{ForEach}\ z\ \texttt{in}\ V\ \texttt{do}\ P \\
 & & \mid & \texttt{While}\ \textit{true}\ \texttt{do}\ \{P; \textit{Click}(n)\} \\
\textit{DOMNodes} & N & ::= & \textit{a list of DOM Nodes expressed in XPath} \\
\textit{EntryData} & V & ::= & \textit{a list of user-provided input data}
\end{array}
$$

Figure 4: The DSL for Web Automation [17]

The synthesizer can generate web automation scripts with rich structures and operations, as defined in the domain-specific language (DSL) in Figure 4. It supports various operations, e.g., clicking a button on the webpage, scraping text or URL, going back to the previous page, and entering data in a text field on a webpage. Note that the data may either come from an input file (i.e., *SendData*) or a constant string typed in by the user (i.e., *SendKeys*). The program locates a DOM node n on the webpage using standard XPath selectors. The DSL can express three types of loops. The first kind

Statements	Translation rules	Examples
`Click` X	"*Click*" + `Text`(X)	"*Click next page*"
`ScrapeText` X	"*Get the text of*" + `Column_name`(X)	"*Get the text of phone number*"
`ScrapeLink` X	"*Get the link of*" + `Column_name`(X)	"*Get the link of store*"
`GoBack`	"*Return to the previous page*"	"*Return to the previous page*"
`SendKeys` Y X	"*Enter*" + `Text`(X) + "to" +`Text`(Y)	"*Enter Iphone 12 to input box*"
`SendData` v0 X	"*Send an input to*" + `Text`(X)	"*Send an input to search box*"
`ForEach` v0 in X	"*Find a list of matched elements (hover to see the matched elements in the target webpage). For each element,*" + `Translate`(statement.body)	"*Find a list of matched elements (hover to see the matched elements in the target webpage). For each element, get the text of phone number*
`While`	"*Repeat the previous step(s)*"	"*Repeat the previous step(s)*"

Table 1: Translation rules for web automation scripts

of loop iterates over a list N of DOM nodes on the webpage. This is typically used for scraping a list of items (e.g., loop over a list of Amazon products on a webpage and extract the price for each product). The next loop type supports iteration over a list V of entries in the input data, where its loop body takes each entry z to fill some field on the webpage. Finally, the while loop repeatedly clicks the "next page" button (located by n) and performs actions on each page using P.

We extended the original synthesis algorithm to support fine-grained user control in MIWA, e.g., adding or deleting an action in the middle of the trace. The original algorithm assumes that new actions can only be appended to the end of the trace. Thus, any edits to the demonstration trace will lead to a re-synthesis from scratch. To improve the efficiency of synthesizing from updated traces, we introduced a new incremental synthesis algorithm with caching. It caches the worklist of program candidates during synthesis. When the user edits an action in the middle of the demonstration trace, the syntheser retrieves the prior state from the cache and continues synthesis from that point instead of starting from scratch. This algorithm greatly improves synthesis efficiency.

5.2 Natural Language Explanation

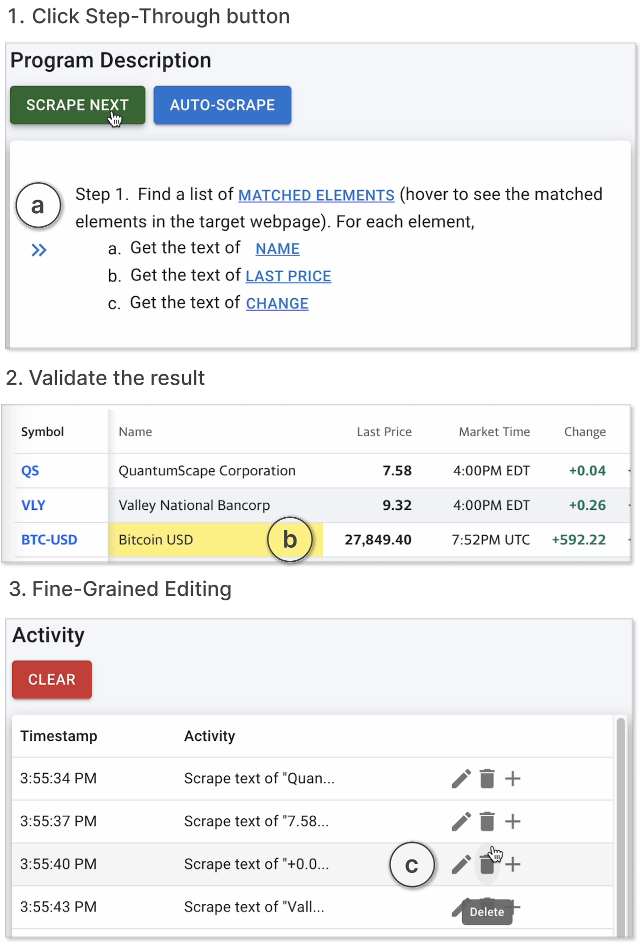
Prior work [36, 42] generates NL explanations for programs based on pattern substitution and templates. Thus, they cannot handle arbitrarily complex programs in the wild. To address this limitation, we propose a new grammar-based method that systematically generates NL explanations by decomposing a script based on its abstract syntax tree (AST) and translating each statement based on a set of composable rules. This divide-and-conquer mechanism allows our method to handle arbitrarily complex programs, including those with nested loops. We describe the details below.

Step-by-Step Natural Language Description: Given a web automation script, our method first parses it into an Abstract Syntax Tree (AST) and identifies different kinds of program statements in it. Then, our method sequentially translates each statement to generate the full explanation. Specifically, we design translation rules for each type of program statement in the DSL, as shown in Table 1. For instance, given a statement `Click` X in a script, MIWA translates the statement based on a text template—"*Click*" + `Text`(X). In this template, `Text`(X) is a auxiliary function that takes an XPath X as input and returns the text value of the HTML element selected by X. If the XPath points to a button with the text "Next Page", then `Text`(X) will return "Next Page" to compose the final string—"*Click Next Page*".

Another auxiliary function used by our translation method is `Column_name`(X). This function is used to translate the `ScrapeText` and `ScrapeLink` statements in a script, as shown in Table 1. Unlike the `Text` function, `Column_name` is hard to be resolved directly from the corresponding HTML elements specified by the XPath X, since the HTML elements typically contain concrete values to be scraped instead of a high-level description. To address this challenge, MIWA leverages user-provided column names in the output table as the description of the scraped data in the column. Given an XPath X, this function tracks which column the scraped data of this XPath is placed to in the output table and retrieves the column name.

Finally, for structured program statements such as loops, our method recursively translates the inner statements to their natural language descriptions and composes them together based on the translation rules.

Visual Correspondence: To enhance the readability of generated NL descriptions, we use XPath to establish a visual correspondence between the description and elements on the target website (**D2**). Specifically, the XPath of an entity mentioned in a program statement is used to create a listener and a clickable link in the React UI. When hovering over these links, the extension will use the XPath to locate and highlight the corresponding target to provide live feedback. Unlike a previous approach [16], which highlights web elements to help users identify scrapable elements, our feature serves a different purpose—*helping users recognize the correspondence between entities in the NL description and elements in the target website.* This is critical, since some entities may be described using terminologies that users are not familiar with and some elements may be ambiguous due to similar elements in the target website.



1. Click Step-Through button

Program Description

SCRAPE NEXT AUTO-SCRAPE

Step 1. Find a list of MATCHED ELEMENTS (hover to see the matched elements in the target webpage). For each element,

a. Get the text of NAME

b. Get the text of LAST PRICE

c. Get the text of CHANGE

2. Validate the result

Symbol	Name	Last Price	Market Time	Change
QS	QuantumScape Corporation	7.58	4:00PM EDT	+0.04
VLY	Valley National Bancorp	9.32	4:00PM EDT	+0.26
BTC-USD	Bitcoin USD	27,849.40	7:52PM UTC	+592.22

3. Fine-Grained Editing

Activity

CLEAR

Timestamp	Activity
3:55:34 PM	Scrape text of "Quan...
3:55:37 PM	Scrape text of "7.58...
3:55:40 PM	Scrape text of "+0.0...
3:55:43 PM	Scrape text of "Vall...

Figure 5: In step-through debugging, a cursor is displayed next to the statement under execution (Figure 5ⓐ). After users click the "Scrape Next"next button, the corresponding elements on the target website will be highlighted to help users validate the program (Figure 5ⓑ). Users can further refine the script by undoing, redoing, or editing the demonstration trace (Figure 5ⓒ). MIWA will generate a new script based on the new demonstration trace.

5.3 Supporting Fine-grained Control

MIWA allows users to validate a generated script with intermediate feedback on each step of execution (**D2**) and to refine the script to fix errors (**D4**). This is supported by the following two features.

Step-through Debugging for Quick Validation and Diagnosis: First, MIWA supports step-through debugging to help users accurately identify which step introduces an error while inspecting the intermediate execution results (Figure 5ⓐ). Specifically, a user can click on the "SCRAPE NEXT" button to scrape one item at a time. The corresponding elements will be highlighted to help users validate the script (Figure 5ⓑ). To implement this feature, we extended the script interpreter to introduce a debug mode, where the interpreter executes a single statement at a time and waits for the signal from the front-end interface to execute the next statement.

Undo, redo and edit demonstration traces. MIWA allows users to undo, redo, or edit an action in the demonstration trace to refine their demonstrations or fix a demonstration mistake (Figure 5ⓒ). MIWA will then regenerate the script based on the updated demonstration trace. By reading the NL description of the newly generated script, users can quickly decide whether the refinement is effective or not. This *Demo-Check-Edit* procedure will continue until a desired script is generated.

5.4 Anomaly detection

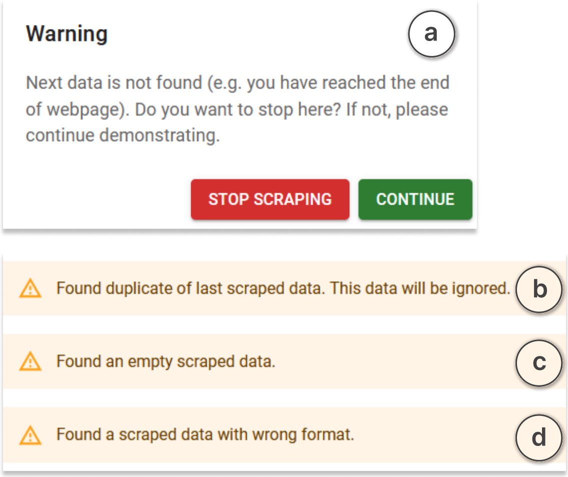
A web automation script may encounter various issues, such as missing values, due to inconsistent web design and unhandled corner cases. Therefore, MIWA employs a set of heuristics to constantly monitor and detect anomalous data during web automation (**D3**). These heuristics are designed based on 29 real-world web automation tasks found in an online forum for web automation, iMacros [5]. These tasks are detailed in Section 8. Overall, MIWA detects four types of data anomalies:

- **Type I** (Invalid action): The web element to scrape or a text field to fill in does not exist on the target website.
- **Type II** (Duplicate action): The web element to scrape is identical to the last scraped data.
- **Type III** (Empty data): The web element to scrape exists but does not contain any content, e.g., text, link.
- **Type IV** (Inconsistent data format): The format of the data to scrape is inconsistent with other data in the same column in the output table.

Specifically, to check data format consistency, we design a set of regular expressions to infer the data format, as shown in Table 2. These regular expressions are deliberately designed so that data instances in different formats will match at most one regular expression, except for the `String` format, which is the default format for all data. For instance, a string "https://www.subway.com" will only match the `Link` format, in addition to the default `String` format.

<table>
<tr><th>Data Type</th><th>Regular Expression Rules</th><th>Examples</th></tr>
<tr><td>String</td><td>^.+$</td><td>Any non-empty string</td></tr>
<tr><td>Text</td><td>^[\d\sA-Za-z,.;:\/\-"'?!~&()\[\]]+$</td><td>Cambridge, MA 02143, USA</td></tr>
<tr><td>Number</td><td>^(?![\s,.+\-]*$)(?:|[+\-])[\d\s,]*\d(?:|[,.][\d\s]+)(?:|%)$</td><td>-20.0%</td></tr>
<tr><td>Link</td><td>^http(?:|s):\/{2}\S+$</td><td>https://www.subway.com</td></tr>
<tr><td>Email</td><td>^\S+@\S+$</td><td>example@domain.com</td></tr>
<tr><td>Date</td><td>^(?:\d{1,2}|\d{4})([.\/\-])\d{1,2}\1(?:\d{1,2}|\d{4})$</td><td>2022/8/31</td></tr>
<tr><td>Money</td><td>^[$€£]\s*(?![\s,.]*$)[\d\s,]*\d(?:|[,.][\d\s]+)$</td><td>$4.02</td></tr>
<tr><td>Phone</td><td>^(?!\d{10})(?:|(?:|\+)\d{1,3})(?:|[\s-])(?:(\d{3})|\d{3})(?:|[\s-])\d{3}(?:|[\s-])\d{4}$</td><td>(765) 838-0054</td></tr>
</table>

Table 2: Regular expressions to infer data format.


Warning

Next data is not found (e.g. you have reached the end of webpage). Do you want to stop here? If not, please continue demonstrating.

STOP SCRAPING CONTINUE

Found duplicate of last scraped data. This data will be ignored.

Found an empty scraped data.

Found a scraped data with wrong format.

Figure 6: Different types of anomaly alerts.

When a data anomaly is detected, a prompt will be displayed to inform the user. For Type I anomalies, the user can either end the scraping session right away or continue demonstrating manually through the pop-up dialog, as shown in Figure 6ⓐ. For Type II, III, and IV anomalies, a non-obstructive alert will be displayed next to the scraped data table for the user's reference when correcting previous actions, as shown in Figure 6ⓑ, ⓒ, and ⓓ.

6 USER STUDY

To evaluate the usefulness and usability of MIWA, we conducted a within-subjects user study with 24 participants who have different levels of programming experience. We chose Rousillon [16], a state-of-the-art web automation system, as a comparison baseline.

6.1 Participants

We envision that MIWA will not only be helpful for non-experts but also improve the productivity of experienced programmers by eliminating the need to manually write code. Thus, in the user study, we recruited participants with different levels of expertise in web automation. Specifically, we recruited 24 students (6 female, 17 male, 1 non-binary) from the Computer Science department at Purdue University via the department mailing list. 14 participants said they had written web automation scripts before, while 10 participants had no experience with web automation. Furthermore, 12 participants had more than 5 years of programming experience, 10 had 2 to 5 years, and 2 had only 1 year. As compensation for their participation, each participant received a $25 Amazon gift card.

6.2 Tasks

We selected three tasks with different levels of difficulty from an existing benchmark [17]. This benchmark consists of real-world web automation tasks from the iMacros forum [5]. The first task is considered easy to solve, and both tools can solve it quickly in one iteration. The second task involves multi-page scraping, requiring users to demonstrate how to navigate to the next page. The third task is considered the hardest, since it requires entering search keywords from a given input file and repeatedly scraping data for different search results. Each task is described below.

Task 1 (Scrape Stock Prices): Open https://finance.yahoo.com/most-active, a financial website that records various company names, stock prices, and changes. You must scrape the contents of the columns *Name*, *Price*, and *Change* for the 25 most active stocks.

Task 2 (Scrape Nearby Garages): Open the garage finder website at https://www.themotorombudsman.org/garage-finder. From this website, you will need to scrape each *garage title*, *hyperlink*, and the *garage's type* on the first page, then click the Next Page button and continue to scrape the data in the following pages.

Task 3 (Scrape Attorney Information): Open https://apps.calbar.ca.gov/attorney/LicenseeSearch/QuickSearch, which is a government website that allows users to search for attorneys. For this task, you need to scrape the attorney's *Number* and *City* based on a given input file that has a list of attorneys' names.

6.3 Protocols

Each user study starts with an introduction and a consent solicitation. Then, a participant was assigned two web automation tasks, one to be completed with MIWA and the other with Rousillon [16]. To mitigate the learning effect, both task and tool assignment orders were counterbalanced across participants. In total, 8 participants experienced each task in each condition. Before starting each task, participants watched a tutorial video of the assigned synthesizer and spent about 5 minutes becoming familiar with the tool. Then, they were given 20 minutes to complete the assigned task. A task was considered failed if participants did not reach a script that could correctly scrape all data after 20 minutes.

After completing each task, participants filled out a post-task survey to give feedback. The post-task survey asked users what they liked or disliked about the assigned tool and what they wished they had. The survey also included a set of Likert-scale questions to ask users to rate the usefulness of key features in each assigned tool. To evaluate the cognitive load of using a tool, we included five NASA Task Load Index questions [18] as part of the post-task survey. In the end, participants filled out a final survey where they directly compared the two tools. We recorded each user study with the participants' permission. A study took an average of 53 minutes.

7 RESULTS

7.1 User Performance

Table 3 shows the performance of the participants using MIWA versus Rousillon in terms of task completion time and attempts. Specifically, we consider a participant to have made a *"re-attempt"* when they (1) submitted the wrong script to the experimenter or (2) started over from scratch again. When using MIWA, all 24 participants successfully completed the assigned task. When using Rousillon [16], one participant failed to complete the task.

For all three tasks, MIWA significantly reduced the task completion time. MIWA's average task completion time is 3 minutes 58 seconds, while Rousillon's completion time is 8 minutes 44 seconds. The mean difference of 4 minutes 46 seconds is statistically significant (unpaired t-test : $t = 4.20551$, $df = 23$, $p = .00006$). Even for the easy task, where both MIWA and Rousillon generate the correct

	Task 1		Task 2		Task 3		Overall	
	MIWA	**Rousillon**	**MIWA**	**Rousillon**	**MIWA**	**Rousillon**	**MIWA**	**Rousillon**
Completion Time	2:16	4:16	4:45	10:10	4:54	11:47	3:58	8:44
# of Attempts	1.25	1.37	1.12	2.37	1.62	2.12	1.33	1.95
# of Trace Refinement	0.87	-	2.5	-	3.75	-	2.37	-

Table 3: The average task completion time and the average number of attempts made by participants.

script in the first round of synthesis, MIWA still saves the task completion time by almost half. Though we did not experiment with a manual programming condition, a previous user study [16] shows that participants took more than 50 minutes to complete a web automation task when using Selenium to manually write web automation scripts. This implies that MIWA can significantly improve user productivity compared with manual programming.

When using MIWA, participants made an average of 1.33 attempts, while they made an average of 1.95 attempts when using Rousillon. The mean difference of 0.62 is statistically significant (unpaired t-test: $p=0.003143$). Participants made an average of 2.37 trace refinement actions to refine their script, e.g., undoing, redoing, and editing previous actions in the demonstration trace.

To understand why MIWA helped participants save so much time, we manually analyzed the post-task survey responses and video recordings. First, we found that the NL description considerably accelerated the process of program comprehension. According to the recordings, all 24 participants relied heavily on the NL description to comprehend the generated script when using MIWA. P14 stated, "*The program description is very user friendly to understand.*" P7 stated, "*It has a simple GUI, offers natural language descriptions and is also very intuitive as I can add elements in order and check them.*" In the post-task survey, 20 participants agreed or highly agreed that NL descriptions helped them comprehend the synthesis code, as shown in Figure 9. In contrast, users of Rousillon spent more time comprehending the Scratch-like program representation. Participants using Rousillon also expressed less confidence on the generated scripts. P17 said, "*It's hard to understand. Using the graphical programming interface is not very different from writing programs. I need to spend lots of time on understand the components. When the scraping is complicated, it's almost similar to read a program.*" P5 said, "*Even if you are a programmer, if you don't have web basics, you will be confused by a column name like list_1_item.*"

Secondly, we found that it was common for participants to make mistakes in previous demonstrations. With MIWA, participants can easily undo, redo, and modify their previous actions in a demonstration trace. Specifically, MIWA users performed an average of 2.37 undo, redo, and editing actions to refine their demonstration traces. However, with Rousillon, participants had to start over again or directly edit the generated script, both of which turned out to be time-consuming. Specifically, when using Rousillon, 15 out participants got stuck due to program editing errors and restarted the application to try again, adding to their total completion time. They took an average of 4 minutes and 31 seconds to finish the tasks after the first failure. By contrast, when using MIWA, only 8 participants failed on the first attempt. During the first attempt, participants encountered 5 **Type I** errors and 2 **Type IV** errors. These errors were promptly noticed and communicated to the participants. On average, they only needed 2 minutes and 8 seconds to refine their traces and successfully complete the task on the second attempt. P21 said, "*I think the edit and delete options are really user-friendly because users do not need to start over when they accidentally click the wrong items or feed the wrong items to the program.*"

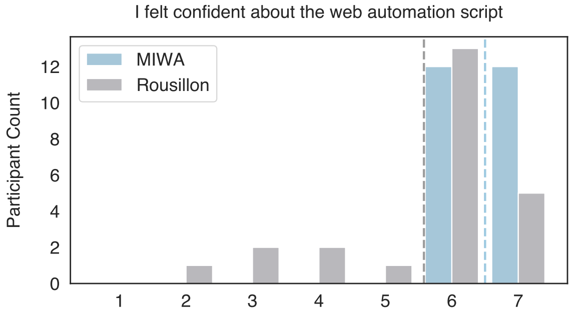


Figure 7: User confidence on the web automation scripts when using Rousillon and MIWA. The dotted lines represent the mean confidence (5.58 vs. 6.50).

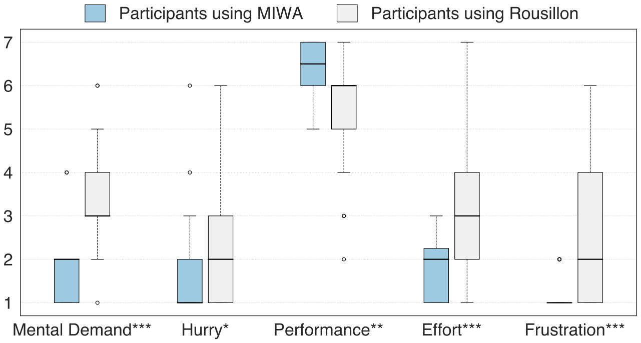


Figure 8: User responses to the NASA TLX questionnaire with statistically significant mean differences marked with asterisks (*: $p<0.05$, **: $p<0.01$, ***: $p<0.001$ based on t-test).

7.2 User Confidence and Cognitive Overhead

In the post-study survey, participants self-reported their confidence in the generated web automation scripts on a 7-point scale (1—very

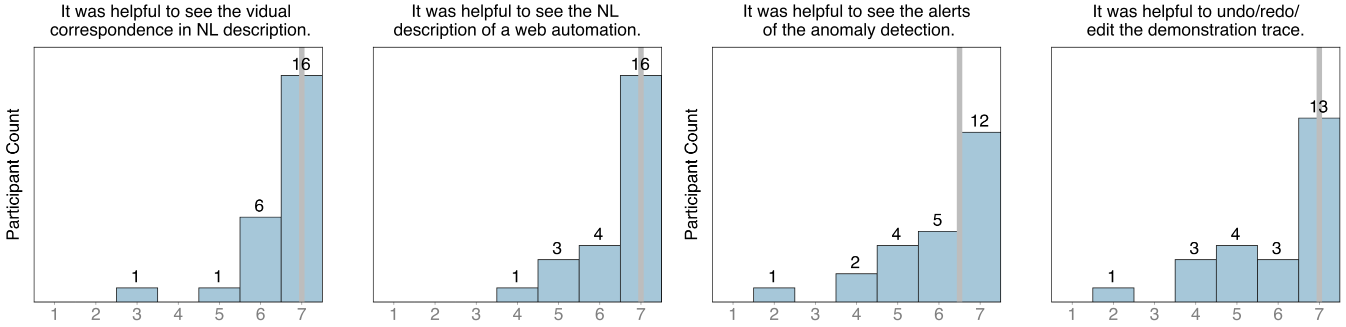


Figure 9: User ratings on the usefulness of individual features (vertical bar indicating the median)

low confidence, 7—very high confidence). As shown in Figure 7, participants expressed a higher level of confidence when using MIWA compared with using Rousillon (6.50 vs. 5.58). The mean difference of 0.92 is statistically significant (unpaired t-test: t = -3.05085, df = 23, p-value = 0.00189). The increase in user confidence may be attributed to NL explanations and visual correspondences provided by MIWA. P17 said, "*I can validate the results by looking at the natural language explanation and the operations list easily.*" P19 said, "*I was able to understand what parts of the webpage were being scraped due to the highlighting feature so I was confident that all the information was correct.*"

As shown in Figure 8, participants using MIWA had less mental strain, effort, and stress. There are three reasons. First, MIWA provides an NL description of the synthesis program, which is intuitive. Therefore, participants were not required to understand the grammar of the web automation language. P14 stated, "*The program description is more intuitive and user-friendly to just check which fields are being recorded.*" Secondly, the visual correspondence feature of MIWA allows users to highlight the matching column on the target page. P18 explained, "*it's helpful to highlight the generated script to see what elements MIWA was considering while scraping.*" Thirdly, users can easily validate the behavior of the generated script using the debugging feature provided by MIWA. P8 said, "*I liked that it was easy to delete mistakes with the scraping, you didn't have to restart the whole program.*"

7.3 User Ratings of Individual Features

Figure 9 shows the participants' ratings over the key features in MIWA. Among all features, the visual correspondence feature is valued the most by participants, followed by the NL description. As P1 explained, "*Highlighting the text we are about to scrape helps a lot. Showing real-time natural language description makes it feel like we are going in the right direction, and we can see it learning and improving.*" Anomaly detection is the third favorite feature with 17 participants confirming its usefulness. P10 said, "*MIWA provides very user-friendly suggestions and warnings.*" Finally, 16 participants found it helpful to undo, redo, and edit their previous actions in the demonstration trace. P4 stated, "*It is very easy to modify or remove the inputs provided.*"

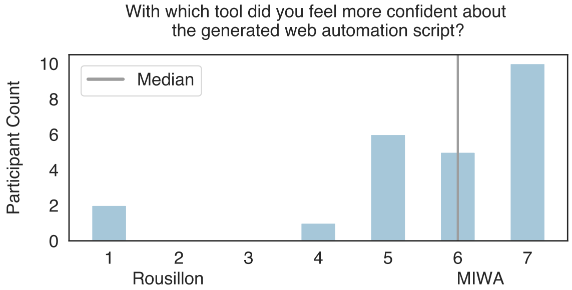


Figure 10: User Preference between Rousillon and MIWA

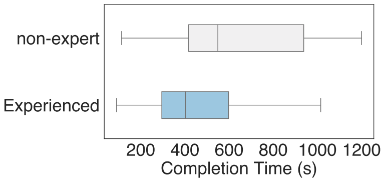


Figure 11: Completion Time Comparison

7.4 User Preference and Feedback

Figure 10 shows the distribution of user preferences between MIWA and Rousillon. 21 out of the 24 participants preferred MIWA over Rousillon. We coded the responses of participants to the question regarding what they liked about MIWA and discovered three themes. First, participants noted that MIWA made the program easy to comprehend. P24 stated, "*It's more easy to understand and validate my operations correctness.*" P5 said, "*You don't need certain background to complete the task.*" Second, participants noted that MIWA assisted them in validating the outcomes of the synthesis. P20 said, "*The executions are editable, there are NL explanations, and it's faster.*" P8 stated, "*I found it more helpful because I was able to go back in my steps if I had made a mistake. The program was also a little easier to understand in terms of what it identified as the steps.*"

In the post-task survey, we asked what additional features or information could have helped participants complete the work more effectively. 4 participants complained about the speed of the synthesizer and hoped for its improvement. 3 participants mentioned that it would be beneficial to have more advanced functions, such as scraping images.

7.5 The Impact of User Expertise

Figure 11 shows the distribution of task completion time for participants who had experience with web automation (i.e., *experienced users*, N=14) vs. participants who had no experience with web automation (i.e., *non-experts*, N=10). Overall, experienced users took an average of 7 minutes and 26 seconds, whereas non-experts took 10 minutes and 34 seconds. However, the mean difference of 3 minutes and 8 seconds was not statistically significant (unpaired t-test: p=0.07019). This implies user expertise appears to have a limited impact on user performance. We can interpret this as a narrowing of users' performance gap between novices and experts, which demonstrates the effectiveness of MIWA.

While all non-experts (10/10) and the majority of experts (11/14) preferred MIWA over Rousillon, three experts found Rousillon more helpful for programmers since they can directly edit the automation script in the visual language. This provides more flexibility and convenience for them. P15 said, "*MIWA is more suitable for people who do not know programming, but Rousillon is more useful for programmers.*" We also did a Chi-Square test of independence and found that there was no statistically significant association between user preference and expertise (p=0.2416). This implies that user expertise has a limited impact on user preference. Nevertheless, it is worthwhile investigating how to support experts in exerting their knowledge and expertise.

8 QUANTITATIVE EVALUATION

We conducted a case study with 29 real-world web automation tasks found in an online forum for web automation [4]. Some examples of these tasks include scraping movie information from IMDb's Most Popular Movies page (benchmark ID 56), scraping product data from multiple pages (benchmark ID 34) from eBay, and scraping company data from government websites based on business names (benchmark ID 127). We have provided the task description, scraping results, and screenshots of the final script synthesized by MIWA for each task in the Supplementary Material.

Since the goal of this quantitative study is to verify that MIWA can be used to solve various tasks instead of evaluating its learnability or usability, the first three authors independently solved these 29 tasks using MIWA. They represent expert users who are familiar with the tool and have rich experience in web automation. Thus, the results of this study should be interpreted as the performance of MIWA in ideal situations. A task is considered failed if an author cannot solve the task after 20 minutes.

Overall, two authors solved 22 of the 29 tasks with an average of 3 minutes and 6 seconds and 2 minutes 58 seconds respectively, and one author solved 21 tasks with an average of 2 minutes and 25 seconds. 20 tasks were solved by all three authors. This result suggests that MIWA may be a useful tool for solving a wide range of web automation tasks.

There are two main reasons why MIWA fails to synthesize the correct scripts for some tasks. First, MIWA ranks the generated scripts by their length and only returns the shortest script to users. Thus, the synthesis algorithm can accidentally leave out correct scripts if they are not the shortest. This limitation can be addressed by rendering top-k scripts for users to choose from. Second, some tasks are too complicated to be solved within 20 minutes. For example, some tasks require generating automation scripts with more than three levels of loops. As the expected number of loops in a script increases, the search space will increase exponentially. This significantly slows down the script generation process.

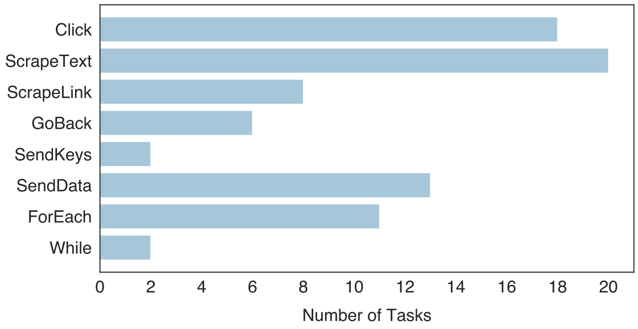


Figure 12: The number of web automation tasks that cover different kinds of statements in the script

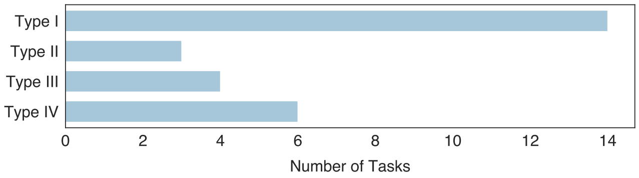


Figure 13: Utility rate of the anomaly detection feature

The results of the quantitative study also indicate that certain translation rules are much more commonly used than others. Specifically, Figure 12 shows how often each rule is triggered in these 29 tasks. The translation rules for `Click` and `ScrapeText` are the most utilized, triggered by 18 and 20 tasks, respectively. The translation rules for `SendKeys` and `While` are the least utilized.

In addition, to investigate the utility rate of the anomaly detection feature, we analyzed the data anomalies that occurred in the quantitative study. Figure 13 shows that the most common data anomaly was **Type I** (invalid action), which occurred in 14 out of 29 tasks. **Type II** (duplicate action) was triggered in 3 out of 29 tasks, while **Type III** (empty data) and **Type IV** (inconsistent data format) were triggered in 4 and 6 out of 29 tasks, respectively. These results show that the anomaly detection feature is frequently triggered in web automation tasks and can cover a range of errors.

9 DISCUSSION

9.1 Design Implications

The results of our user study indicate that providing interaction support for program understanding and validation in PBD-based web automation can significantly increase both task-solving efficiency and user confidence in the generated script. Therefore, it is worth continuing to investigate new interactive mechanisms to facilitate program comprehension and validation in web automation. Here, we conclude our design implications:

9.1.1 Enhancing User Understanding through Step-by-step Natural Language (NL) Explanation. Future work on web automation and PBD should consider incorporating natural language explanations to aid users in program comprehension and validation. This feature is highly appreciated by our users. Not only does this make the automation script more accessible to users, but it also significantly simplifies the process of identifying and resolving any bugs in the web automation scripts. Thus, NL explanations would serve as an efficient communication vehicle for both non-experts and experts, bridging the gap between complex programs and user mental models.

9.1.2 Supporting program debugging and repairing for end-users. Providing debug and repairing mechanisms that are friendly to end-users is critical in the realm of web automation. During demonstrations, it is common for participants to make mistakes while demonstrating. MIWA addresses this issue by enabling users to step through each individual step in an NL description. This feature allows them to observe live feedback and directly edit the demonstration trace to fix observed errors. While the comparison baseline allows users to directly edit the generated script to fix errors, our participants found it difficult to use since they are not familiar with the visual programming language used by the baseline.

9.1.3 Detecting Inconsistencies in Web Structures and Elements. In the web automation process, it is common to encounter inconsistent web elements on different pages of the same website, or even across similar websites. These inconsistencies can lead to automation errors and, in severe cases, system crashes. Therefore, it is important to detect these inconsistencies, promptly communicate them to users, and provide support to help users handle them. This would not only prevent potential system breakdowns but also enhance the overall automation process.

9.1.4 Improving Synthesis Efficiency in Complex Tasks. Synthesis efficiency remains a significant challenge in PBD systems, especially in complex tasks. Several potential solutions could be considered to address this issue. For instance, researchers might explore the possibility of synthesizing partial programs, a technique that breaks down complex tasks into smaller, manageable sub-tasks, thereby improving synthesis efficiency. Furthermore, providing users with guidance on how to decompose tasks could also prove beneficial. Lastly, implementing mechanisms to solicit early feedback on task performance could help identify and rectify issues sooner, thus enhancing the overall efficiency of the process. Another promising solution is to predict the script even if the users did not finish the whole demonstration, which could save time and prevent potential demonstration errors. If the system can propose the possible program in advance, the user only needs to validate the program without further demonstrations. This feature would be similar to how the code autocompletion feature works in Copilot [2].

9.2 Limitations

The quantitative evaluation (Section 8) reveals several technical limitations of MIWA. First, only returning the top-1 script generated by a PBD system is likely to miss correct but lower-ranked scripts. Therefore, it would be helpful to provide multiple alternative scripts for users to select from. Second, MIWA currently does not support some dynamic webpage features such as dynamically rendered contents (e.g., infinite scrolling list). Third, MIWA uses a simple heuristic-based method to detect data anomalies. Thus, there may be corner cases that cannot be handled by our current heuristics. Besides, MIWA currently does not support file downloading due to safety concerns, but it allows users to scrape links of the files to download via the ScrapeLink operator. MIWA does not support timing, which is critical for handling asynchronous events and other time-related tasks.

Our current user study design restricts us to attributing participants' success to individual MIWA features separately. Though we asked participants to rate the usefulness of each feature in the post-task survey (Section 7.3), this feature is a subjective measurement. A more objective way to measure the usefulness of each feature is to create variants of the system by disabling each key feature and use them as comparison baselines in the user study.

9.3 Future Work

9.3.1 Render Multiple Synthesized Programs in PBD Systems. Our synthesis algorithm currently ranks generated scripts based on their length and always returns the shortest script. This strategy sometimes misses correct programs that are not the shortest. It can be improved in two ways. First, PBD systems should allow users to navigate alternative program candidates rather than just a single one. To mitigate the cognitive overhead caused by inspecting multiple programs, it is necessary to investigate effective and interactive mechanisms to enable swift and seamless navigation and comparison among multiple programs. Second, it is worthwhile investigating more effective algorithms to rank and select synthesized programs to render. For instance, one can experiment with multi-criteria ranking [56] to select a small set of representative but diverse program candidates. There is also prior work that uses machine learning to select programs [54, 59].

9.3.2 Eliciting Human Feedback to Guide Program Synthesis for Complex Tasks. It takes a long time to synthesize a script for complex tasks. A promising solution is to synthesize a partial program first and elicit human feedback to concretize the partial program. Specifically, user actions that are not supported or generalizable can be represented as "holes" in the script. During script execution, when encountering a hole, the PBD system will pause and prompt users to perform the actions manually. Once it is completed, the automated execution will resume and in the meantime, the synthesizer will make another attempt to synthesize an expression to fill the hole based on user actions. This method allows for the seamless integration of manual actions into the otherwise automated process, increasing the system's versatility and generalizability.

9.3.3 Enhancing Internationalization. MIWA's synthesis component can be applied to non-English websites, as it relies on the DOM structure rather than the content of elements. However, to fully support internationalization, we need to adjust the translation rules outlined in Table 1 to enable the translation of a synthesized script into different languages. Furthermore, we need to update the regular expressions in Table 2 to identify data anomalies in a non-English context.

10 CONCLUSION

In this paper, we introduce MIWA, an interactive PBD system that automatically generates scripts for web automation. To help users gain more trust in the generated script, MIWA provides a step-by-step natural language description of the script with visual correspondence to the content on the target website. MIWA also supports fine-grained control over the web automation process by allowing users to undo, redo, and edit the demonstration trace. Users can validate the program step by step without having to restart the tool. In addition, MIWA automatically detects web anomalies in order to capture the inherent noise in user demonstrations. A user study of 24 participants found that when using MIWA, participants completed assigned tasks in less than half the time and with more confidence compared to using a state-of-the-art PBD-based web automation tool [16].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by the National Science Foundation under grant numbers CCF-2236233 and CCF-2123654.

REFERENCES

[1] 2022. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup

[2] 2022. Copilot. https://github.com/features/copilot

[3] 2022. Cypress. https://www.cypress.io

[4] 2022. iMacros. https://www.progress.com/imacros

[5] 2022. iMacros.net Forum. https://forum.imacros.net/

[6] 2022. Mechanical Soup. https://mechanicalsoup.readthedocs.io/en/stable

[7] 2022. Puppeteer. https://pptr.dev

[8] 2022. Scrapy. https://scrapy.org

[9] 2022. Selenium. https://www.selenium.dev

[10] 2022. UiPath. https://www.uipath.com

[11] Christine Alvarado and Randall Davis. 2007. Resolving ambiguities to create a natural computer-based sketching environment. In *ACM SIGGRAPH 2007 courses*. 16–es.

[12] Ashley Amaya, Ruben Bach, Florian Keusch, and Frauke Kreuter. 2021. New data sources in social science research: Things to know before working with Reddit data. *Social science computer review* 39, 5 (2021), 943–960.

[13] Kimberly A Barchard and Larry A Pace. 2011. Preventing human error: The impact of data entry methods on data accuracy and statistical results. *Computers in Human Behavior* 27, 5 (2011), 1834–1839.

[14] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. 2015. Browser record and replay as a building block for end-user web automation tools. In *Proceedings of the 24th International Conference on World Wide Web*. 179–182.

[15] Sarah Elizabeth Chasins. 2019. *Democratizing Web Automation: Programming for Social Scientists and Other Domain Experts*. Ph. D. Dissertation. UC Berkeley.

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

[17] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: web robotic process automation using interactive programming-by-demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 152–167.

[18] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[19] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R Klemmer. 2007. Programming by a sample: rapidly creating web applications with d. mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 241–250.

[20] Andrea Hess, Karin Anna Hummel, Wilfried N Gansterer, and Günter Haring. 2015. Data-driven human mobility modeling: a survey and engineering guidance for mobile networking. *ACM Computing Surveys (CSUR)* 48, 3 (2015), 1–39.

[21] Chris Hess and Sarah E Chasins. 2022. 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*. 1–9.

[22] David F Huynh, Robert C Miller, and David R Karger. 2006. Enabling web browsers to augment web sites' filtering and sorting functionalities. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*. 125–134.

[23] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J Brostow. 2019. Hilc: domain-independent pbd system via computer vision and follow-up questions. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 9, 2-3 (2019), 1–27.

[24] Yaochu Jin, Handing Wang, Tinkle Chugh, Dan Guo, and Kaisa Miettinen. 2018. Data-driven evolutionary optimization: An overview and case studies. *IEEE Transactions on Evolutionary Computation* 23, 3 (2018), 442–458.

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

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

[27] Brittany Kondo and Christopher Collins. 2014. Dimpvis: Exploring time-varying information visualizations by direct manipulation. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2003–2012.

[28] Raymond Kosala and Hendrik Blockeel. 2000. Web mining research: A survey. *ACM Sigkdd Explorations Newsletter* 2, 1 (2000), 1–15.

[29] Rebecca Krosnick and Steve Oney. 2021. Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9.

[30] Rebecca Krosnick and Steve Oney. 2022. ParamMacros: Creating UI Automation Leveraging End-User Natural Language Parameterization. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10.

[31] Vlad Krotov and Matthew Tennyson. 2018. Research note: scraping financial data from the web using the R language. *Journal of Emerging Technologies in Accounting* 15, 1 (2018), 169–181.

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

[33] Jürgen Landauer and Masahito Hirakawa. 1995. Visual AWK: a model for text processing by demonstration. In *Proceedings of Symposium on Visual Languages*. IEEE, 267–274.

[34] Tessa Lau et al. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*. 65–67.

[35] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

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

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

[38] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. 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)*. IEEE, 105–114.

[39] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. 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*. 577–589.

[40] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. 2009. End-user programming of mashups with vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*. 97–106.

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

[42] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[43] Tyler H McCormick, Hedwig Lee, Nina Cesare, Ali Shojaie, and Emma S Spiro. 2017. Using Twitter for demographic and social science research: tools for data collection and processing. *Sociological methods & research* 46, 3 (2017), 390–421.

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

[45] Jan Meskens, Kris Luyten, and Karin Coninx. 2010. 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* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 129–138. https://doi.org/10.1145/1866029.1866051

[46] Robert C Miller and Brad A Myers. 2002. LAPIS: Smart editing with text structure. In *CHI'02 Extended Abstracts on Human Factors in Computing Systems*. 496–497.

[47] Francesmary Modugno and Brad A. Myers. 1997. Visual programming in a visual shell—A unified approach. *Journal of Visual Languages & Computing* 8, 5-6 (1997), 491–522.

[48] Mauro Mosconi and Marco Porta. 2000. Iteration constructs in data-flow visual programming languages. *Computer languages* 26, 2-4 (2000), 67–104.

[49] Brad A Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 534–541.

[50] Brad A Myers and William Buxton. 1986. Creating highly-interactive and graphical user interfaces by demonstration. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 249–258.

[51] Brad A Myers, Jade Goldstein, and Matthew A Goldberg. 1994. Creating charts by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 106–111.

[52] Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. 2022. SemanticOn: Specifying Content-Based Semantic Conditions for Web Automation Programs. (2022).

[53] Yury Puzis. 2012. An interface agent for non-visual, accessible web automation. In *Adjunct proceedings of the 25th annual ACM symposium on User interface software and technology*. 55–58.

[54] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[55] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[56] Dan Shen, Jie Zhang, Jian Su, Guodong Zhou, and Chew Lim Tan. 2004. Multi-criteria-based active learning for named entity recognition. In *Proceedings of the 42nd annual meeting of the Association for Computational Linguistics (ACL-04)*. 589–596.

[57] Ben Shneiderman. 1981. Direct manipulation: A step beyond programming languages. In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems.(Part-II): Human Interface and the User Interface-Volume 1981*. 143.

[58] Ben Shneiderman. 1982. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology* 1, 3 (1982), 237–256.

[59] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*. Springer, 398–414.

[60] Arjun Srinivasan and John Stasko. 2017. Orko: Facilitating multimodal interaction for visual exploration and analysis of networks. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 511–521.

[61] Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.

[62] Elior Vila, Galia Novakova, and Diana Todorova. 2017. Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats. In *Proceedings of the International Conference on Advances in Image Processing*. 144–150.

[63] Ashenafi Zebene Woldaregay, Eirik Årsand, Ståle Walderhaug, David Albers, Lena Mamykina, Taxiarchis Botsis, and Gunnar Hartvigsen. 2019. Data-driven modeling and prediction of blood glucose dynamics: Machine learning applications in type 1 diabetes. *Artificial intelligence in medicine* 98 (2019), 109–134.

[64] Jeffrey Wong and Jason I Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1435–1444.

[65] Tal Yarkoni, Dean Eckles, James AJ Heathers, Margaret C Levenstein, Paul E Smaldino, and Julia I Lane. 2021. Enhancing and accelerating social science via automation: Challenges and opportunities. *Harvard Data Science Review* 3, 2 (2021).