ParamMacros: Creating UI Automation Leveraging End-User Natural Language Parameterization

Rebecca Krosnick
Computer Science and Engineering
University of Michigan
Ann Arbor, MI USA
rkros@umich.edu

Steve Oney
School of Information
University of Michigan
Ann Arbor, MI USA
soney@umich.edu

***Abstract*—Prior work in programming-by-demonstration (PBD) has explored ways to enable end-users to create custom automation without needing to write code. We propose a new end-user specification model – asking the end-user to explicitly identify parts of their natural language query that can be generalized. We built a PBD system, ParamMacros, where users first generalize a concrete natural language question – identifying parameters and their possible values – and then create a demonstration of how to answer the question on the website of interest. ParamMacros then infers a generalized program by using the user-provided parameter values to identify relevant patterns in the website's structure. In a lab study we found that participants were able to meaningfully parameterize natural language queries and felt such a parameterization and demonstration process would be useful for creating custom automation.**

***Index Terms*—parameterization, natural language, programming-by-demonstration, automation, virtual assistants**

I. 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 [1] 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) [2], [3] is an effective way to allow users—including users without programming experience—to create user interface (UI) automation macros [4]–[8]. 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 concrete 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 [4] and Appinite [5] 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) [9] 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

This work is supported by NSF Award 2007857.

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

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.

II. Related Work

A. Virtual Assistants

Virtual assistants like Siri [1], Alexa [10], Google Assistant [11], and Cortana [12] 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.

B. Writing UI Automation Scripts

Developers can write custom automation scripts that programmatically mimic a user's interactions on a user interface (UI). Popular frameworks include Selenium [13], Puppeteer [14], Cypress [15], and Beautiful Soup [16] for the web, and Shortcuts [17] and App Actions [18] for mobile. However, writing such scripts is non-trivial – for example, it can be challenging to construct UI selectors [19] that are robust across different inputs [20]. Record and replay tools like Selenium IDE [21] and Cypress Studio [22] 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 enables users to create generalized macros without writing code.

C. Programming by Demonstration

Programming by demonstration (PBD) [2], [3], enables end-users to create computer programs without writing code – instead users just provide concrete demonstrations or examples of the desired behavior. A key challenge of PBD is inferring user intent and generalizing from demonstrations. PBD has a rich history, with systems that support UI creation [23], [24], text and code editing [25]–[27], data transformation [28], [29], constructing regular expressions [30], [31], and more.

Prior work has explored using PBD for creating web scraping scripts. Rousillon [32] and WebRobot [33] can synthesize nested loop-based scraping logic from user demonstrations by leveraging patterns in the DOM [9]. Our work similarly leverages patterns in the DOM, but we focus on generating parameterized programs that use user-provided parameters.

Task automation is another domain with a rich PBD history. CoScripter [34], [35] 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. Our work uses parameters to generalize dynamic element selection.

Most similar to our work is the Sugilite suite [4]–[6], which enables end-users to create custom automation for responding to speech requests and completing tasks on their mobile device. With Sugilite [4], 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 [5] 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 [6] 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.

AutoVCI [8] and VASTA [7] 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.

Etna [36] 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 [37] generates task shortcuts for user NL requests – it maps a user's NL request to the best-matching app screen from the Rico dataset [38] 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.

D. 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 [39], 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 [40] explores a semantic parser approach to map a user's natural language to the most relevant high-level conceptual action on the given website. ParamMacros leverages built-in heuristics and user-provided custom demonstrations rather than models trained on large datasets.

E. Question-Answering Systems

Question-answering systems [41] 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.

F. Natural Language and Data

Natural language interfaces to databases (NLIDBs) [42] 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. Our work helps end-users create custom automation on-demand when there is no database already.

Prior work also explores natural language interfaces for data visualizations [43]–[45]. DataTone [45] is an NLIDB that allows flexibility for ambiguious 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 [46] 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 our work we ask users to identify parameters themselves.

III. 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 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 to create automation macros to answer these kinds of questions from data on the Major League Baseball (MLB) statistics web page[1][2].

A. 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 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. ParamMacros proposes potential **parameter values** (Figure 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 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.

B. 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 IV). 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 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

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

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

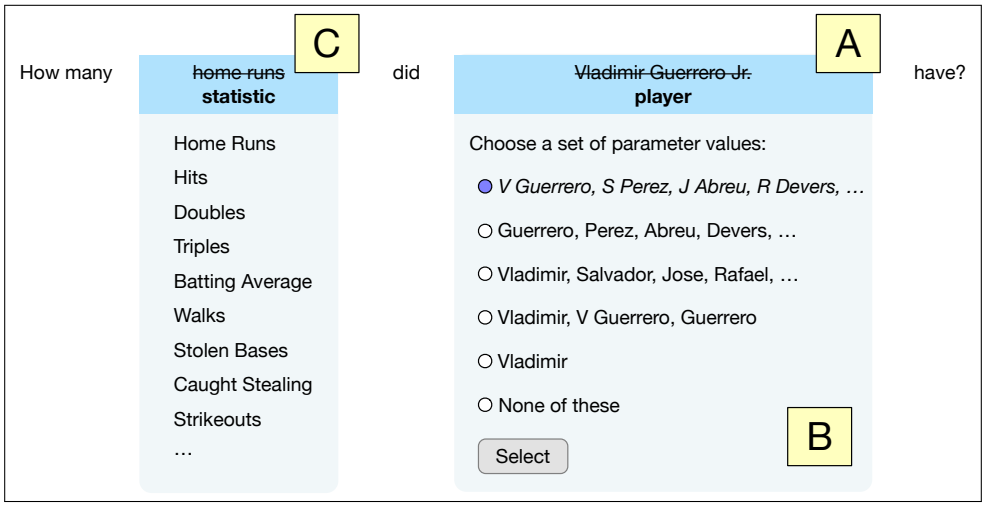

Fig. 1. An illustration of ParamMacros's UI for parameterizing natural language queries. The user has chosen to (A) generalize "Vladimir Guerrero Jr." to make the parameter *player* and (C) generalize "home runs" to parameter *statistic*. The system proposes possible alternative values (B) for each parameter for the user to select from.

for the correct answer (the "HR"—short for Home Runs[3]—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.

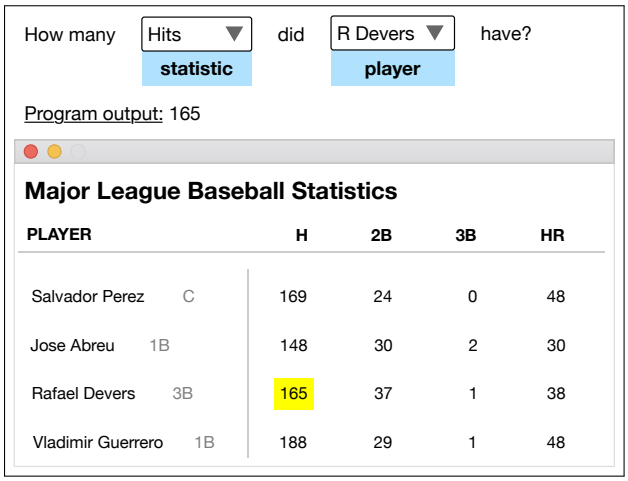

Fig. 2. An illustration of ParamMacros's execution interface and the Major League Baseball statistics website. When the user runs the generated macro with the inputs *<statistic>=Hits* and *<player>=R Devers*, it returns and highlights the correct answer, 165.

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 2).

[3]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".

C. 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 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 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 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>*).

D. 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

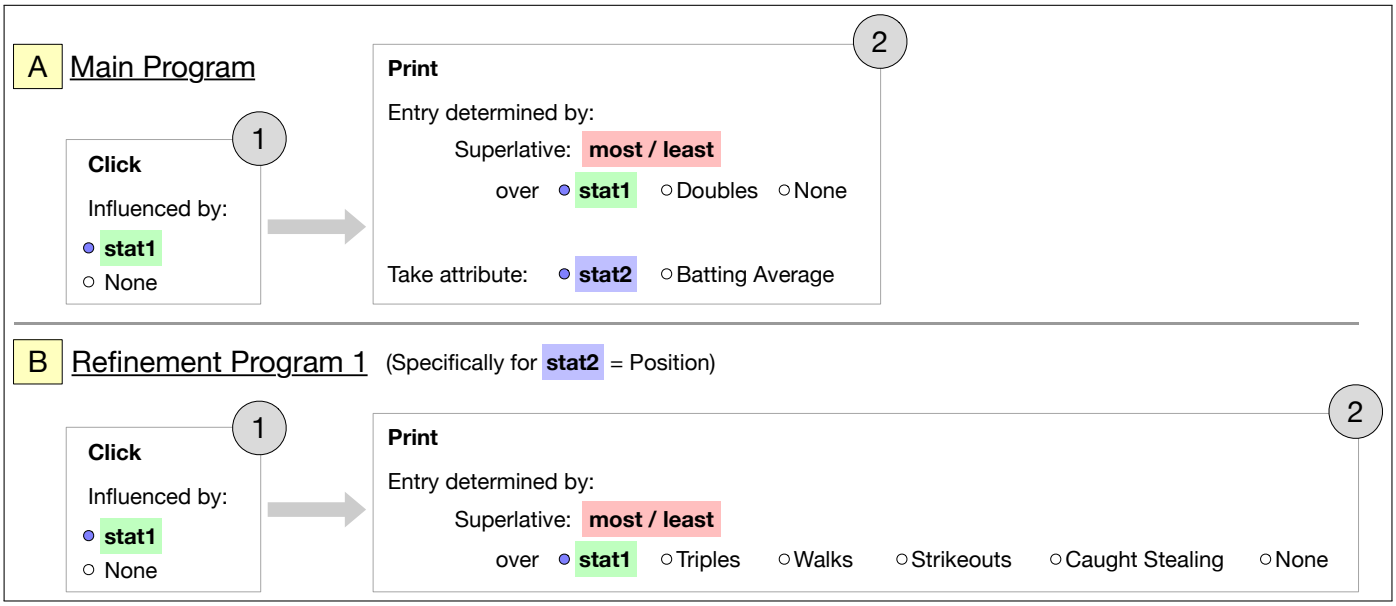

Fig. 3. Program description for "For the player who had the <*most/least*> <*stat1*>, what was their <*stat2*>?" The program (1) first clicks a header in the statistics table to sort the data, and then (2) prints out a value from the table. (A) General program logic used for all parameter input values except <*stat2*> = *Position*. (B) Logic generated from the user's refinement demonstration; used only when the user runs the program with <*stat2*> = *Position*.

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 IV).

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

IV. 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.

A. Parameter values

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

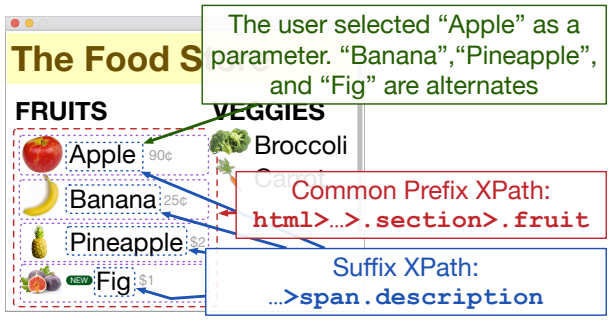

Fig. 4. An explanatory illustration of our inference algorithm on an imaginary website titled "The Food Store". Here, the user has selected "Apple" as a parameter in their NL query and wants "Banana", "Pineapple", and "Fig" to be alternative values. Our algorithm infers a common suffix across candidate parameter values and a common suffix across target elements.

parameter values. For example, if the user asks "How much does one <Apple> cost?" on the page in Figure 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[4][5] 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, a unique path for "Apple" might be `html >...> .section[1] > .fruit[1] > div[1] > span[2]`. A key insight of our algorithm is that

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

[5]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.

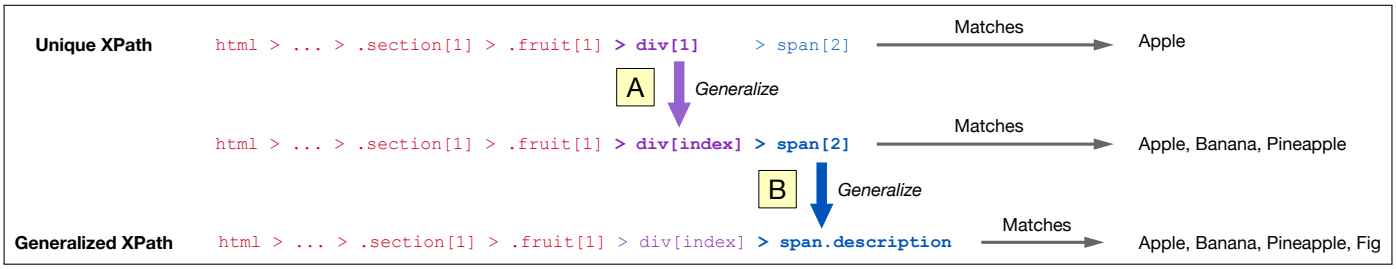

Fig. 5. The process to transform a single value's XPath to a generalized XPath formula that works across parameter values. The algorithm starts with a unique XPath matching the original parameter value, "Apple". (A) The algorithm then identifies possible "iteration points" that generate alternative parameter 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".

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 5A). There are many possible iteration points for a given XPath query. In the above query for Figure 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.

B. 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 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– 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, our algorithm would therefore find the more general suffix `> span.description` (Figure 5B).

C. 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.

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 IV-D), 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).

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, 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 IV-B. 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, the algorithm generalizes the target XPath suffix to be `> span.price`.

D. Limitations

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.

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 [6] 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.

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[7], 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) 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).

V. 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.

A. 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

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

[7]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

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.

B. Study Design

Our user study involved two meaningfully different sites: the Forbes billionares list[6] and an IMDb movie list[7]. 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:

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.

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)[8]. This allowed us to see variety in how people parameterize different queries, as well as observe patterns for a common query.

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?".

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.

VI. 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.

A. Parameterizing questions

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[8] 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[8] 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.

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).

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

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

actually means" (P10) but that they better understood after seeing parameters applied later in the program creation stage.

B. 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.

C. 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.

D. 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.

VII. 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 [5] 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 [47].

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 [6].

VIII. 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.

Acknowledgments

We thank our study participants for their time and effort, and our anonymous reviewers for their feedback which has helped

improve the paper. We also thank members of the University of Michigan HCI community for their feedback.

REFERENCES

[1] "Siri," https://www.apple.com/siri/, accessed: 2022-04-06.

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

[3] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.

[4] T. J.-J. Li, A. Azaria, and B. A. Myers, "Sugilite: creating multimodal smartphone automation by demonstration," in *Proceedings of the 2017 CHI conference on human factors in computing systems*, 2017, pp. 6038–6049.

[5] T. J.-J. Li, I. Labutov, X. N. Li, X. Zhang, W. Shi, W. Ding, T. M. Mitchell, and B. 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)*. IEEE, 2018, pp. 105–114.

[6] T. J.-J. Li, M. Radensky, J. Jia, K. Singarajah, T. M. Mitchell, and B. 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*, 2019, pp. 577–589.

[7] A. R. Sereshkeh, G. Leung, K. Perumal, C. Phillips, M. Zhang, A. Fazly, and I. Mohomed, "Vasta: a vision and language-assisted smartphone task automation system," in *Proceedings of the 25th international conference on intelligent user interfaces*, 2020, pp. 22–32.

[8] L. Pan, C. Yu, J. Li, T. Huang, X. Bi, and Y. Shi, "Automatically generating and improving voice command interface from operation sequences on smartphones," in *CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–21.

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

[10] "Amazon alexa voice ai," https://developer.amazon.com/en-US/alexa, accessed: 2022-04-06.

[11] "Google assistant," https://assistant.google.com/, accessed: 2022-04-06.

[12] "Cortana," https://www.microsoft.com/en-us/cortana, accessed: 2022-04-06.

[13] "Selenium," https://www.selenium.dev/, accessed: 2020-09-11.

[14] "Puppeteer," https://pptr.dev/, accessed: 2020-09-18.

[15] "Cypress," https://www.cypress.io/, accessed: 2021-03-19.

[16] "Beautiful soup," https://www.crummy.com/software/BeautifulSoup/, accessed: 2022-06-05.

[17] "Shortcuts user guide," https://support.apple.com/guide/shortcuts/welcome/ios, accessed: 2022-06-06.

[18] "App actions overview," https://developers.google.com/assistant/app/overview, accessed: 2022-06-06.

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

[20] R. Krosnick and S. Oney, "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, 2021, pp. 1–9.

[21] "Selenium ide," https://www.selenium.dev/selenium-ide/, accessed: 2020-06-08.

[22] "Cypress studio," https://docs.cypress.io/guides/core-concepts/cypress-studio, accessed: 2021-05-05.

[23] R. G. McDaniel and B. A. Myers, "Getting more out of programming-by-demonstration," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1999, pp. 442–449.

[24] M. R. Frank, P. N. Sukaviriya, and J. 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*, 1995, pp. 167–175.

[25] T. A. Lau, P. M. Domingos, and D. S. Weld, "Version space algebra and its application to programming by demonstration." in *ICML*. Citeseer, 2000, pp. 527–534.

[26] R. C. Miller and B. A. Myers, "Interactive simultaneous editing of multiple text regions." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 161–174.

[27] W. Ni, J. Sunshine, V. Le, S. Gulwani, and T. 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*, 2021, pp. 258–269.

[28] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011.

[29] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. 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*, 2020, pp. 1–12.

[30] A. F. Blackwell, "Swyn: A visual representation for regular expressions," in *Your wish is my command*. Elsevier, 2001, pp. 245–XIII.

[31] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 627–648.

[32] S. E. Chasins, M. Mueller, and R. Bodik, "Rousillon: Scraping distributed hierarchical web data," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 963–975.

[33] R. Dong, Z. Huang, I. I. Lam, Y. Chen, and X. 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.

[34] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 1719–1728.

[35] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 943–946.

[36] O. Riva and J. Kace, "Etna: Harvesting action graphs from websites," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 312–331.

[37] D. Arsan, A. Zaidi, A. Sagar, and R. Kumar, "App-based task shortcuts for virtual assistants," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 1089–1099.

[38] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. 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*, 2017, pp. 845–854.

[39] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, "Mapping natural language instructions to mobile ui action sequences," in *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.

[40] S. Mazumder and O. 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.

[41] F. Zhu, W. Lei, C. Wang, J. Zheng, S. Poria, and T.-S. Chua, "Retrieving and reading: A comprehensive survey on open-domain question answering," *arXiv preprint arXiv:2101.00774*, 2021.

[42] K. Affolter, K. Stockinger, and A. Bernstein, "A comparative survey of recent natural language interfaces for databases," *The VLDB Journal*, vol. 28, no. 5, pp. 793–819, 2019.

[43] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A natural language interface for visual analysis," in *Proceedings of the 29th annual symposium on user interface software and technology*, 2016, pp. 365–377.

[44] V. Setlur, E. Hoque, D. H. Kim, and A. 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*, 2020, pp. 966–978.

[45] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. 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*, 2015, pp. 489–500.

[46] Z. Chen and H. Xia, "Crossdata: Leveraging text-data connections for authoring data documents," in *CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–15.

[47] G. Litt and D. Jackson, "Wildcard: spreadsheet-driven customization of web applications," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, 2020, pp. 126–135.