WebRobot: Web Robotic Process Automation using Interactive Programming-by-Demonstration

Rui Dong
University of Michigan

Zhicheng Huang
University of Michigan

Ian Iong Lam
University of Michigan

Yan Chen
University of Toronto

Xinyu Wang
University of Michigan

Abstract

It is imperative to democratize robotic process automation (RPA), as RPA has become a main driver of the digital transformation but is still technically very demanding to construct, especially for non-experts. In this paper, we study how to automate an important class of RPA tasks, dubbed *web RPA*, which are concerned with constructing software bots that automate interactions across data and a web browser. Our main contributions are twofold. First, we develop a formal foundation which allows semantically reasoning about web RPA programs and formulate its synthesis problem in a principled manner. Second, we propose a web RPA program synthesis algorithm based on a new idea called *speculative rewriting*. This leads to a novel *speculate-and-validate* methodology in the context of rewrite-based program synthesis, which has also shown to be both theoretically simple and practically efficient for synthesizing programs from demonstrations. We have built these ideas in a new interactive synthesizer called WebRobot and evaluate it on 76 web RPA benchmarks. Our results show that WebRobot automated a majority of them effectively. Furthermore, we show that WebRobot compares favorably with a conventional rewrite-based synthesis baseline implemented using egg. Finally, we conduct a small user study demonstrating WebRobot is also usable.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Keywords: Program Synthesis, Programming by Demonstration, Rewrite-based Synthesis, Robotic Process Automation, Web Automation, Human-in-the-loop Systems

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523711

ACM Reference Format:
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 (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523711

1 Introduction

Robotic process automation (RPA) is a software technology that aims to streamline the process of creating *software robots* that emulate user interactions with digital applications such as web browsers and spreadsheets [3, 5, 9, 24, 33, 34, 61, 64]. These robots are essentially programs: like humans, they can perform tasks such as entering data, completing keystrokes, navigating across pages, extracting data, *etc.* However, they, once programmed, can perform tasks much faster with fewer mistakes. Therefore, RPA has the potential to significantly simplify business workflows and improve the productivity for *both organizations and individuals* [7]. Gartner predicted that RPA will remain the fastest-growing software market in the next several years [49].

While RPA has become a main driver of the digital transformation, it is still technically very demanding to construct automation programs, and consequently, not everyone can build software robots that suit their needs. For instance, it is estimated that 3-7% of tasks deemed important *by an organization* have been automated, whereas a long tail of more than 40% of *individual-driven* tasks yet are still to be automated [7]. These tasks represent a high percentage of automation opportunities to scale RPA to individual non-expert end-users.

Web RPA. How to democratize RPA in order to foster its adoption among non-experts is a broad, new but increasingly important problem. In this paper, we consider an important subset of RPA tasks, dubbed *web RPA*, and investigate how to automate this class of tasks. As illustrated in Figure 1, web RPA involves *interactions between data and a web browser*. For example, it involves *programmatically* entering data (in a semi-structured format), extracting data from webpages, as well as navigating across multiple webpages. Conceptually, web RPA is close to web/browser automation, where the key distinction is that RPA emphasizes *interactions across/within*

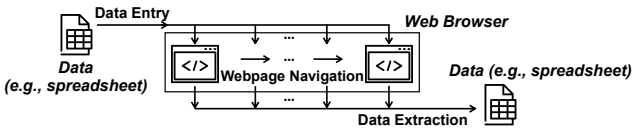

Figure 1. Illustration of web RPA.

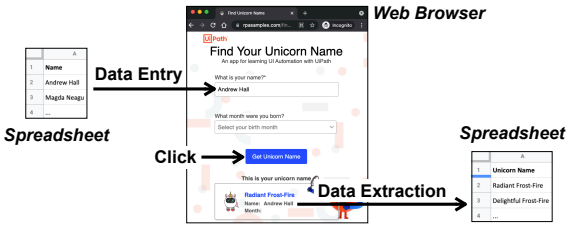

Figure 2. A real-world web RPA problem from UiPath.

applications, while browser automation has to do with *web browsers*. In other words, one can view web RPA as the "intersection" of RPA and web automation; hence the name.[1]

Let us consider a real-world scenario (shown in Figure 2) from a recent webinar [6] by UiPath, a leading RPA company. In this example, a manager working for a unicorn adoption agency wanted to test a hypothesis that sending follow-up emails with their unicorn names to customers would increase the adoption rate. Unfortunately, the customer relationship management system is disconnected from the web-based unicorn name generator. In other words, there is no easy way to automatically generate a unicorn name for each customer. The manager tried to seek help from the IT but was told that creating an automation program for this job is expensive unless there is a provably sufficient return on it. In the end, they had to manually perform this experiment: export customer information into a spreadsheet, *copy-paste every name from the sheet, enter it in the unicorn generator, scrape the generated name for each customer,* and finally send emails with unicorn names. This is very tedious. A key problem in this process is how to create a program that interacts with the web-based generator and the spreadsheet in order to create names for all customers. This is exactly a web RPA problem.[2]

Web RPA sits at the intersection of multiple areas, such as programming languages and human-computer interaction. While it has been studied in different forms by different communities, to the best of our knowledge, there is no principled approach that automatically generates web RPA programs in a comprehensive manner. For instance, while being able to scrape data across webpages, Helena [17] has relatively less support for programmatic data entry. Furthermore, it may generate wrong programs which, in our experience, are not always easy to "correct" using Helena's build-in features. On the other hand, the HCI and databases communities have

[1]Web automation is a broad term. We note that web RPA is highly related to web automation but in this work, we do not precisely distinguish them.
[2]This example involves one single webpage but we have many benchmarks that involve navigating across multiple pages (see Section 7).

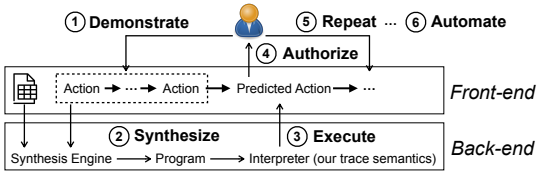

Figure 3. Schematic workflow of our approach.

proposed various interfaces [35, 37] and wrapper induction techniques [10, 25, 50], which are even more restricted and can automate only single-webpage tasks. Finally, while some "low-code" solutions based on record-and-replay exist on the industrial market (such as iMacros [2]), they require significant manual efforts (e.g., adding loops), which makes them potentially less accessible to non-expert end-users.

Interactive programming-by-demonstration (PBD) for web RPA. Our first contribution is a new approach that automates web RPA tasks from demonstrations *interactively*. Compared to existing work, our approach is more automated, resilient to ambiguity, and applicable for web RPA. Figure 3 shows the schematic workflow of our approach. To automate a task, the user just needs to perform it as usual but using our interface (step ①). All the user-demonstrated *actions* are recorded and sent to our back-end synthesis engine. Then, we synthesize a program P that "satisfies" the demonstration (step ②). That is, P is guaranteed to *reproduce* the recorded actions, but P may also *produce more actions afterwards*. We then "execute" P to produce an action that the user *may want to perform next* and visualize this predicted action via our interface (step ③). Finally, the user inspects the prediction and chooses to accept or reject it (step ④). This *interactive* process repeats until there is sufficient confidence that the synthesized program is intended (step ⑤); after that, it will take over and automate the rest of the task (step ⑥). Note that, if at any point the user spots anything abnormal, they can still interrupt and enter the demonstration phase again.

We highlight several salient features of our approach. First, it is *automated*: users only need to provide demonstrations, without needing to write programs. Second, it is *interactive*: whenever a synthesized program is not desired, the user can simply interrupt and continue demonstrating more actions, without having to edit programs. Finally, it could synthesize programs effectively from an expressive language, thanks to *a systematic problem formulation* and *a new search algorithm.*

Systematic formulation of PBD for web RPA. Our second contribution is a systematic formulation for the problem of synthesis from *action-based demonstrations*. In particular, the question we aim to address here is: what does it mean for a program to *satisfy a trace of user-demonstrated actions*? This problem is extremely understudied in a formal context. To the best of our knowledge, the latest work to date is the seminal work [29–31] by Tessa Lau and their co-authors in the 1990s. However, Lau's work considers *state-based* demonstrations; that is, in their work, a demonstration is defined as a sequence of program states. In contrast, our work concerns

action-based demonstrations—a demonstration is a trace of actions. In this context, we are not aware of any prior work that has formalized the *semantic* notion of satisfaction. As a result, existing techniques [17, 18, 39, 40] resort to heuristics and task-specific rules to detect patterns in the action trace in order to generalize it to programs with loops. In this work, we formulate the action-based PBD problem by formalizing the *trace semantics* for an expressive web RPA language. In a nutshell, our semantics "executes" a program (with loops) and produces its "execution trace" of *actions* by unrolling loops and replacing variables with values. Therefore, with our semantics, we can now *check* a program against a trace of actions. Furthermore, this semantics also plays a pivotal role in our *search* algorithm, which we will explain next.

Action-based PBD using speculative rewriting. Once we can check a program against a trace of actions, the next question we ask is: how to *search* for programs that satisfy the given trace? This brings us to the third contribution of our work, which is a novel rewrite-based synthesis algorithm based on a new idea called *speculative rewriting*. The basic idea is simple: we rewrite a slice of the trace into a (one-level) loop which produces that slice, using a set of predefined rules; if we do this iteratively, we can generate nested loops from the inside out. The issue is, it is very hard to define a *complete* set of *correct-by-construction* rules in our domain, because our trace may result from executing loops (from an arbitrary program) for arbitrarily many times. In other words, pattern-matching the *entire* trace in a purely rule-based manner does not scale. In order to scale to complex programs, our idea is to combine *rule-based pattern-matching* and *semantic validation* in the rewrite process via an intermediate *speculation* step. More specifically, instead of pattern-matching *all iterations* to directly generate *true rewrites*, our idea is to pattern-match *a couple of iterations* and generate *speculative rewrites*, or *s-rewrites*. While an s-rewrite might not be a true rewrite in general, they *over-approximate* the set of true rewrites and are much easier to generate. We then use our *trace semantics* to *validate* s-rewrites and retain only those true rewrites.

Our method is closely related to two lines of work. First, it builds upon the "guess-and-check" idea introduced by the counterexample-guided inductive synthesis (CEGIS) framework [54], but we show how to extend this idea for rewrite-based synthesis beyond the traditional application scenarios with example-based and logical specifications. Second, our method incorporates the idea of *semantic rewrite rules* from recent work [42, 62], but we augment this standard correct-by-construction, rule-based rewrite approach with a novel guess-and-check step.[3] We found this new idea to be both theoretically simple and practically efficient in our domain. We also believe this methodology is potentially useful in the more general context of rewrite-based synthesis and in other problem domains with similar trace generalization problems.

[3]We will elaborate on this in the remainder of this paper.

Human-in-the-loop interaction model. As a proof-of-concept, we have also developed a user interface to facilitate user interactions with our synthesizer. Our interface combines programming-by-demonstration, action visualization, and interactive authorization within a human-in-the-loop model, which has shown to be useful in practice for reducing the gulfs of execution and evaluation [44].

Implementation and evaluation. We have implemented our proposed ideas in a tool called WebRobot and evaluate it across four experiments. First, we evaluate WebRobot's synthesis engine on 76 real-world web RPA benchmarks and show that it can synthesize programs effectively. Second, we perform an ablation study and show that all of our proposed ideas are important. Furthermore, we conduct a user study with eight participants which shows that WebRobot can be used by non-experts. Finally, we compare WebRobot with a rewrite-based synthesis approach and our results show that WebRobot significantly advances the state-of-the-art.

In summary, this paper makes the following contributions:

- We identify the web RPA program synthesis problem.
- We formalize a trace semantics of our web RPA language, laying the formal foundation for its synthesis problem.
- We present a novel programming-by-demonstration algorithm based on a new idea called speculative rewriting.
- We develop a new human-in-the-loop interaction model.
- We implement our ideas in a new tool called WebRobot.
- We evaluate WebRobot on 76 tasks and via a user study.

2 Overview of WebRobot

In this section, we highlight some key features of WebRobot using a motivating example[4] from the iMacros forum.

Motivating example. Given a list of zip codes, Ellie wants to extract store information from the Subway website[5]. Since Ellie is not familiar with programming, she has to *manually* perform this task (shown in Figure 4): (a) enter the first zip in the search box, (b) click the search button which then shows five pages of search results, (c) scrape store information on the first page, (d) click the "next page" button and repeat this process for all pages and for all zip codes.

WebRobot. Ellie could use our tool to automate this task. Once WebRobot is fired up, Ellie would first import the list of zip codes and then perform the task using WebRobot. This process is illustrated in Figure 5(a). In particular, Ellie first drags the first zip and drops it in the search bar (action 1). Then, she clicks the GO button and starts scraping information of the first two stores on the first page (actions 2-6). All these actions are recorded by WebRobot in an action trace, which is shown in Figure 5(b). After six actions, WebRobot is able to synthesize a program P_1, as shown in Figure 5(c), which extracts the address and phone number for each store on the first page. Next, WebRobot performs an interactive

[4]https://forum.imacros.net/viewtopic.php?f=7&t=21028

[5]http://www.subway.com/storelocator/

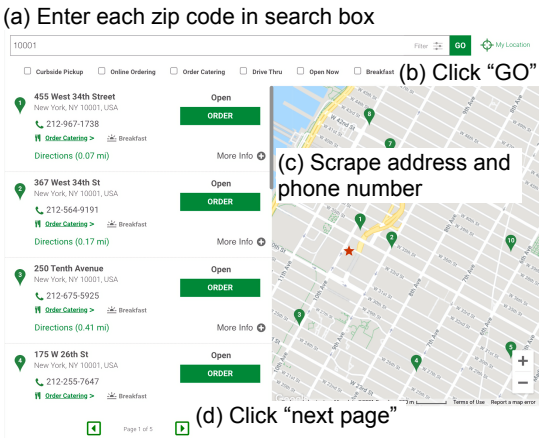

Figure 4. A motivating example: scrape address and phone number for all stores across all pages and for all zip codes.

"authorization" step: it executes P_1 to produce the next action which is then visualized to Ellie (see Figure 5(a), action 7). This is correct, so Ellie accepts it. After a couple of rounds, WebRobot takes over and automates the scraping work on the first page (Figure 5(a), actions 9-22).

WebRobot would terminate after action 22. Thus, Ellie needs to click the "next page" button and extract information for a couple of stores on the second page. These actions are also recorded; see Figure 5(b), actions 23-27. At this point, WebRobot infers a different program P_2 which has two loops one after another, where the second loop extracts information of all stores on the second page. Using P_2, WebRobot is able to automatically scrape the second page; however, it terminates, again, right before the "next page" button.

This time, once Ellie clicks "next page" (i.e., action 44), we can synthesize P_3–see Figure 5(c)–which contains an outer *while* loop that first uses an inner loop for scraping and then clicks "next page" at the end. P_3 now is able to automatically scrape all store information for the remaining pages.

Since Ellie needs to repeat this scraping process for all zip codes, she will enter the second zip code and click "GO" again (actions 107-108), after which WebRobot can synthesize P_4 that has a three-level loop. P_4 first iterates over all zip codes in the given list and then uses a doubly-nested loop to scrape across all pages. At this point, Ellie is done.

We highlight some salient features of WebRobot below.

Interactive PBD to resolve ambiguity. WebRobot does not need users to provide multiple small demonstrations as in traditional PDB approaches [30]; instead, it synthesizes programs *while* the user is performing the task. In case the synthesized program is not intended, WebRobot does not ask the user for an edit on the program as in program-centric tools (such as Helena). Rather, it allows the user to take over and correct the behavior. For instance, in the "authorization" phase, WebRobot visualizes potentially multiple options for the next action and let the user select the one that is desired. This design aims to facilitate interactive disambiguation.

Satisfaction check using trace semantics. Consider P_1 from Figure 5(c) which is synthesized from the first 6 actions $a_1, \cdot\cdot, a_6$ in Figure 5(b). In other words, P_1 *satisfies* the action trace $[a_1, \cdot\cdot, a_6]$. To perform this satisfaction check, we *simulate* the execution of P_1 using our trace semantics. Note that this is a simulated execution, rather than actually executing P_1 in the browser, because during the synthesis process, programs might have side-effects that are not intended. Our trace semantics would first execute the *EnterData* and *Click* statements before the loop, which essentially reproduces a_1, a_2. Then, we unroll the loop twice, reproducing $a_3, \cdot\cdot, a_6$. A subtle aspect here is that the actions produced by our simulated execution might not be *syntactically* the same as those in the recorded trace, since P_1 might use selectors that are different from those in the recorded action trace. Thus, we check if an action produced by our semantics and an action in the demonstrated trace refer to the same Document Object Model (DOM) node; this is done by also recording a trace of DOMs in tandem. We will explain this in detail in Section 3.

Rewrite-based PBD. How to synthesize programs from an action trace? We take a rewrite-based approach. Consider the trace $[a_1, \cdot\cdot, a_6]$ and P_1 in our previous example: the loop in P_1 is rewritten from actions $a_3, \cdot\cdot, a_6$. WebRobot can also synthesize nested loops. For instance, the inner loop in P_3 corresponds to multiple *slices* of actions, such as 3-22, 24-43. Once identified, these slices are rewritten to (multiple occurrences of) the same loop. Then, WebRobot will generate a nested loop from the inside out by essentially treating the (inner) loop as one action and rewriting again. In this case, it rewrites actions 3-106 to the *while* loop in P_3.

Selector search. In addition to identifying iteration boundaries, WebRobot also considers other selectors, beyond full XPath expressions that are recorded in the trace, since the desired program may not use those recorded. For example, a_4 in Figure 5(b) is a full XPath, whereas the corresponding statement in P_1 (namely, the second statement in the loop) uses a more general selector (with *div[@class='locatorPhone']*). Considering alternative selectors allows to induce more general programs, but it also makes the problem more challenging.

Speculative rewriting. A standard rewrite-based synthesis approach requires a set of *correct-by-construction* rewrite rules [42, 62], meaning they always generate sound rewrites. In our domain, if we follow this idea, we need to design rules that pattern-match actions which result from an unknown number of loop iterations and from arbitrarily complex loop structures; this is hard to scale to complex web RPA tasks. *Our idea is to pattern-match actions from a couple of iterations.* For instance, given $[a_1, \cdot\cdot, a_{22}]$ in Figure 5(b) that corresponds to P_1, instead of pattern-matching $a_3, \cdot\cdot, a_{22}$, using rules, to synthesize a *true* rewrite (i.e., a loop), we pattern-match $a_3, \cdot\cdot, a_6$ and *speculate* a potential rewrite P_l, assuming $a_3, \cdot\cdot, a_6$ "come from" the first two iterations of P_l. This is conceptually simpler and faster, but the downside is that P_l might not be a true rewrite, since it is inferred from only the first two iterations.

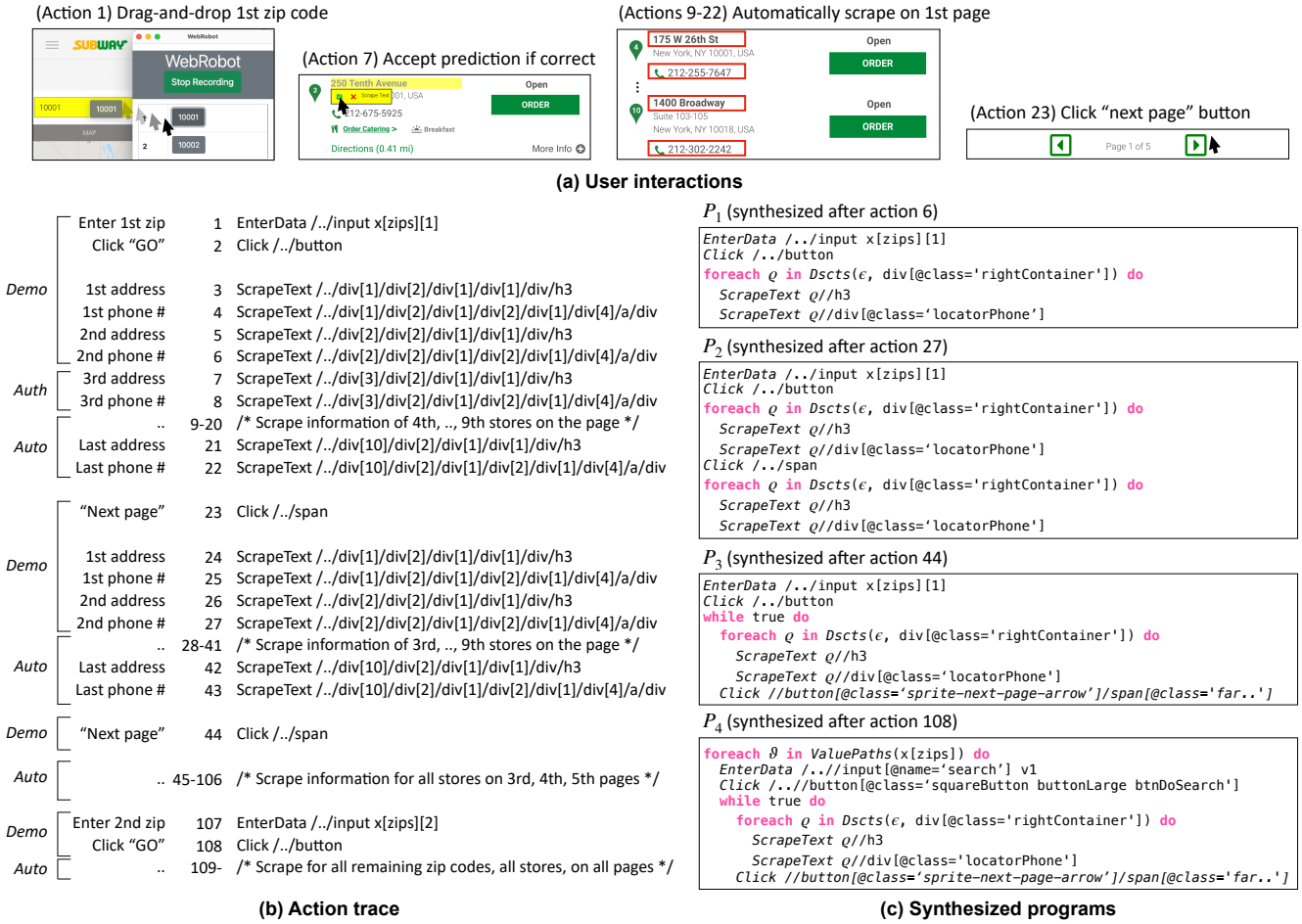

Figure 5. (a) User interactions with the web browser and WebRobot. (b) Action trace recorded by WebRobot, where a short explanation is attached to the left of each action. (c) Programs synthesized by WebRobot at different points.

Semantic validation. Our next idea is to check if a speculative rewrite (or, s-rewrite) P_l is a true rewrite by executing P_l under our trace semantics. The goals are twofold. First, we check if P_l can rewrite a *longer* slice of actions beyond the first iteration; if not, we filter out P_l. Second, if P_l indeed rewrites beyond the first iteration, semantic validation also gives us the (longer) slice that P_l could rewrite. As we can see, building a formal semantic foundation allows us to not only systematically formulate the synthesis problem for web RPA, but also develop an effective algorithm to solving it.

3 Web RPA Language and Trace Semantics

This section lays the formal foundation for web RPA.

3.1 Syntax

Our syntax is shown in Figure 6. Intuitively, a program P in this language is a sequence of statements that emulates user interactions with a web browser and a data source. Its input variable x is a data source I represented in JSON-like format:

$$I \quad ::= \{key : value, \cdots, key : value\}$$
$$key ::= string \qquad value ::= string \mid integer \mid I \mid [value, \cdots, value]$$

This allows using any semi-structured data as our data source.

A statement S, in the simplest case, performs an action on the current webpage. For example, *Click* clicks a DOM node located by a selector n. *ScrapeText* scrapes the text inside a node specified by n. Some statements are parameterless (e.g., *GoBack* that goes back to the previous page and *ExtractURL* that gives the URL of current webpage). Some statements might take multiple parameters. For example, *SendKeys* types a constant string s into an editable field given by a selector n. *EnterData* enters a value v from input data I to a field located by n. Note that v is represented using a *value path*, which is essentially a sequence of keys and array indices in order to access the value from I. On the other hand, a selector n in our language is essentially an XPath expression [8] but it may contain a variable ϱ *at the beginning*. In particular, $n/\phi[i]$ gives the i-th *child* of a DOM node n that satisfies a predicate ϕ. $n//\phi[i]$ gives the i-th *descendant* which satisfies ϕ among all nodes in the subtree rooted at n. Our language has multiple types of predicates. The simplest one is an HTML tag t. For instance, $n/span[1]$ returns the first child of n with tag *span*. The next predicate $t[@\tau = s]$ means the desired DOM node should have tag t and its attribute τ should take value

$$
\begin{array}{llll}
\textit{Program} & P & ::= & S; \cdot\cdot; S \\
\textit{Statement} & S & ::= & \textit{Click}(n) \mid \textit{ScrapeText}(n) \mid \textit{ScrapeLink}(n) \\
 & & \mid & \textit{Download}(n) \mid \textit{GoBack} \mid \textit{ExtractURL} \\
 & & \mid & \textit{SendKeys}(n, s) \mid \textit{EnterData}(n, v) \\
 & & \mid & \texttt{foreach}\ \varrho\ \texttt{in}\ N\ \texttt{do}\ P \qquad \textit{(selectors loop)} \\
 & & \mid & \texttt{foreach}\ \vartheta\ \texttt{in}\ V\ \texttt{do}\ P \qquad \textit{(value path loop)} \\
 & & \mid & \texttt{while}\ \textit{true}\ \texttt{do}\ \{P; \textit{Click}(n)\} \qquad \textit{(while loop)} \\
\textit{Selector} & n & ::= & \epsilon \mid \varrho \mid n/\phi[i] \mid n/\!\!/\phi[i] \\
\textit{Value Path} & v & ::= & x \mid \vartheta \mid v[\textit{key}] \mid v[i] \\
\textit{Selectors} & N & ::= & \textit{Children}(n, \phi) \mid \textit{Dscts}(n, \phi) \\
\textit{Value Paths} & V & ::= & \textit{ValuePaths}(v) \\
\textit{Predicate} & \phi & ::= & t \mid t[@\tau = s]
\end{array}
$$

$s ::= \textit{string} \quad i ::= \textit{integer} \quad t ::= \textit{HTML tag} \quad \tau ::= \textit{HTML attribute}$

Figure 6. Syntax of our web RPA language.

s. For instance, $n/\!\!/ div[@class = \text{“}a\text{”}][2]$ returns the second descendant of n that has tag *div* and whose *class* attribute value is string “a”.

Statements could also be loopy. The first type of loop is *selector loops* which iterate over a list N of DOM nodes on a webpage. This construct is to emulate loopy user interactions on webpages, such as scraping a list of elements. In particular, N returns a list of selectors. During the i-th iteration, the loop variable ϱ binds to the i-th selector in N under which the loop body P gets executed. Note that a statement in P could use ϱ and it may also be a loop. The next loop type is *value path loops*, which are used to emulate loopy interactions with input data. In this case, V evaluates to a list of value paths, ϑ binds to each value path, and P is executed in this context. Our last type of loops is *while loops*, where the termination condition is that the DOM node n in the last *Click* statement no longer exists on the webpage. This construct is primarily used to handle pagination where the user needs to repeatedly click the “next page” button until there is no next page.

3.2 Trace semantics

So far, we have seen the DSL syntax, which is new but fairly standard. Now, in this section, we will formalize its semantics, which is a key distinction of our paper from prior work.

Design rationale. Let us first briefly present our thought process in designing this semantics. Recall that a program P in our language takes as input a data source I and is executed on an initial DOM; P has side-effects that change the DOM, eventually terminating at some browser state. Thus, one may start with the following semantics definition.

$$\pi, \Sigma \vdash P : \pi'$$

Here, π is the initial DOM, Σ is an environment that tracks variable values, and π' is the final DOM when P terminates.

However, there is a gap between this semantics and our specification: one is based on DOMs and the other is based on actions. This brings us to *our first key insight*: the semantics (for our synthesis technique) should incorporate actions that the program executes. This is primarily because in synthesis, we typically use semantics to validate candidate programs against the specification. This leads to the following design.

$$\pi, \Sigma \vdash P : A', \pi'$$

The key idea in this design is to track the trace A' of actions taken by P during its execution. Here, A' is a list $[a_1, \cdot\cdot, a_m]$ of actions where an action is defined as follows.

$$
\begin{array}{l}
a ::= \textit{Click}(\rho) \mid \textit{ScrapeText}(\rho) \mid \textit{ScrapeLink}(\rho) \mid \textit{Download}(\rho) \\
\quad \mid \textit{GoBack} \mid \textit{ExtractURL} \mid \textit{SendKeys}(\rho, s) \mid \textit{EnterData}(\rho, \theta) \\
\rho ::= \epsilon \mid \rho/\phi[i] \mid \rho/\!\!/\phi[i] \qquad \theta ::= x \mid \theta[\textit{key}] \mid \theta[i]
\end{array}
$$

Note that, different from statement S in Figure 6, an action a is loop-free and uses *concrete* selectors ρ and value paths θ.

We further illustrate how action tracking works using the following two simple rules (which include an environment in the output). Other rules are fairly similar.

$$\frac{\pi, \Sigma \vdash S_1 \rightsquigarrow A', \pi', \Sigma' \qquad \pi', \Sigma' \vdash S_2, \cdot\cdot, S_m \rightsquigarrow A'', \pi'', \Sigma''}{\pi, \Sigma \vdash S_1; \cdot\cdot; S_m \rightsquigarrow A' +\!\!+ A'', \pi'', \Sigma''} \ (\textsc{Seq})$$

$$\frac{\Sigma \vdash n \rightsquigarrow \rho \qquad \boxed{\pi' = \texttt{Perform\_Click}(\rho, \pi)}}{\pi, \Sigma \vdash \textit{Click}(n) \rightsquigarrow [\textit{Click}(\rho)], \pi', \Sigma} \ (\textsc{Click})$$

The Seq rule is standard in that it executes S_i's in sequence; however, note that it concatenates the action traces A' and A'' in the output. The actual action tracking takes place in the base rules, such as Click, but there is an issue: Click *actually* performs the operation in the browser. This is problematic because during synthesis, candidate programs might have undesired side-effects (e.g., clicking a button that deletes the database), which prevents us from actually running them.

This motivates *our second key idea*: we *simulate* the actual semantics without actually running P, in particular, by simulating P's DOM transitions using a trace Π of DOMs.[6] We illustrate how this simulation works still on Seq and Click.

$$\frac{\Pi, \Sigma \vdash S_1 \rightsquigarrow A', \Pi', \Sigma' \qquad \Pi', \Sigma' \vdash S_2, \cdot\cdot, S_m \rightsquigarrow A'', \Pi'', \Sigma''}{\Pi, \Sigma \vdash S_1; \cdot\cdot; S_m \rightsquigarrow A' +\!\!+ A'', \Pi'', \Sigma''} \ (\textsc{Seq})$$

$$\frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \qquad \Sigma \vdash n \rightsquigarrow \rho \qquad \boxed{\Pi' = [\pi_2, \cdot\cdot, \pi_m]}}{\Pi, \Sigma \vdash \textit{Click}(n) \rightsquigarrow [\textit{Click}(\rho)], \Pi', \Sigma} \ (\textsc{Click})$$

Here, instead of tracking the resulting DOM π', Click tracks the resulting DOM *trace* Π'; the intuition is that Π' contains DOMs that *future actions* will be executed upon (instead of only the *next immediate action*). The transition from Π to Π' is “angelic” in that Click *always* transitions to the next DOM (by removing π_1 from Π), without actually doing the click on π_1. But, what if performing the click on π_1 does not yield π_2? This is indeed possible, especially given that our synthesis algorithm often explores many wrong programs. However, if the resulting action trace A' matches the user-provided trace A (i.e., the specification), we know that every DOM transition must be genuine, because that is what we had recorded from the user demonstration (assuming deterministic replay). In other words, evaluating “the right” program that corresponds to Π guarantees to yield A' that matches the specification A.

[6] We can obtain this DOM trace by recording intermediate DOMs in tandem while recording the user-demonstrated actions.

$$\text{(Eval)}\quad \frac{\Pi, \{x \mapsto I\} \vdash P \rightsquigarrow A', \Pi', \Sigma'}{\Pi, I \vdash P : A'}$$

$$\text{(Term)}\quad \frac{\Pi = [\,]}{\Pi, \Sigma \vdash P \rightsquigarrow [\,], [\,], \Sigma}$$

$$\text{(Seq)}\quad \frac{\Pi, \Sigma \vdash S_1 \rightsquigarrow A', \Pi', \Sigma' \quad \Pi', \Sigma' \vdash S_2; \cdot\cdot; S_m \rightsquigarrow A'', \Pi'', \Sigma''}{\Pi, \Sigma \vdash S_1; \cdot\cdot; S_m \rightsquigarrow A' +\!\!+ A'', \Pi'', \Sigma''}$$

$$\text{(Click)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash n \rightsquigarrow \rho}{\Pi, \Sigma \vdash \mathit{Click}(n) \rightsquigarrow [\mathit{Click}(\rho)], [\pi_2, \cdot\cdot, \pi_m], \Sigma}$$

$$\text{(EnterData)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash n \rightsquigarrow \rho \quad \Sigma \vdash v \rightsquigarrow \theta}{\Pi, \Sigma \vdash \mathit{EnterData}(n, v) \rightsquigarrow [\mathit{EnterData}(\rho, \theta)], [\pi_2, \cdot\cdot, \pi_m], \Sigma}$$

$$\text{(S-Init)}\quad \frac{\Pi, \Sigma \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N_{\geq 1}\ \texttt{do}\ P \rightsquigarrow A', \Pi', \Sigma'}{\Pi, \Sigma \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N\ \texttt{do}\ P \rightsquigarrow A', \Pi', \Sigma'}$$

$$\text{(S-Cont)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash N_{\geq i} \rightsquigarrow \rho :: N'_{\geq i+1} \quad \mathit{valid}(\rho, \pi_1) \quad \Pi, \Sigma[\varrho \mapsto \rho] \vdash P; \texttt{foreach}\ \varrho\ \texttt{in}\ N'_{\geq i+1}\ \texttt{do}\ P \rightsquigarrow A', \Pi', \Sigma'}{\Pi, \Sigma \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N_{\geq i}\ \texttt{do}\ P \rightsquigarrow A', \Pi', \Sigma'}$$

$$\text{(S-Term)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash N_{\geq i} \rightsquigarrow \rho :: N'_{\geq i+1} \quad \neg \mathit{valid}(\rho, \pi_1)}{\Pi, \Sigma \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N_{\geq i}\ \texttt{do}\ P \rightsquigarrow [\,], \Pi, \Sigma}$$

$$\text{(VP-Loop)}\quad \frac{\Sigma \vdash V \rightsquigarrow [\theta_1, \cdot\cdot, \theta_m] \quad \Pi_0 = \Pi \quad \Sigma_0 = \Sigma \quad \Pi_{i-1}, \Sigma_{i-1}[\vartheta \mapsto \theta_i] \vdash P \rightsquigarrow A_i, \Pi_i, \Sigma_i \quad 1 \leq i \leq m}{\Pi, \Sigma \vdash \texttt{foreach}\ \vartheta\ \texttt{in}\ V\ \texttt{do}\ P \rightsquigarrow A_1 +\!\!+ \cdots +\!\!+ A_m, \Pi_m, \Sigma_m}$$

$$\text{(While-Init)}\quad \frac{\Pi, \Sigma \vdash P; \texttt{if}\ \mathit{valid}(n)\ \texttt{do}\ \{\mathit{Click}(n); \texttt{while}\ \mathit{true}\ \texttt{do}\ \{P; \mathit{Click}(n)\}\} \rightsquigarrow A', \Pi', \Sigma'}{\Pi, \Sigma \vdash \texttt{while}\ \mathit{true}\ \texttt{do}\ \{P; \mathit{Click}(n)\} \rightsquigarrow A', \Pi', \Sigma'}$$

$$\text{(While-Cont)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash n \rightsquigarrow \rho \quad \mathit{valid}(\rho, \pi_1) \quad \Pi, \Sigma \vdash P \rightsquigarrow A', \Pi', \Sigma'}{\Pi, \Sigma \vdash \texttt{if}\ \mathit{valid}(n)\ \texttt{do}\ P \rightsquigarrow A', \Pi', \Sigma'}$$

$$\text{(While-Term)}\quad \frac{\Pi = [\pi_1, \cdot\cdot, \pi_m] \quad \Sigma \vdash n \rightsquigarrow \rho \quad \neg \mathit{valid}(\rho, \pi_1)}{\Pi, \Sigma \vdash \texttt{if}\ \mathit{valid}(n)\ \texttt{do}\ P \rightsquigarrow [\,], \Pi, \Sigma}$$

Figure 7. Trace semantics of our web RPA language.

Our trace semantics. Let us now explain our simulation semantics in detail. Our top-level judgment takes the form:

$$\Pi, I \vdash P : A'$$

where A' is the action trace produced by P, and Π is used to guide the simulated execution (as we also briefly discussed earlier). Our key rule is of the form:

$$\Pi, \Sigma \vdash P \rightsquigarrow A', \Pi', \Sigma'$$

which intuitively states:

> *Given DOM trace Π and environment Σ, P would execute actions in A', yielding environment Σ' and DOM trace Π' (containing DOMs future actions will be executed upon).*

Evaluating programs. The Term rule states that, if the input DOM trace is empty (i.e., there is no DOM to execute P upon), then we terminate the entire execution. Otherwise, we evaluate the statements sequentially using the Seq rule.

Evaluating loop-free statements. Figure 7 gives two example rules; the other rules are very similar. The Click rule first evaluates n to obtain a concrete selector ρ and then produces a *Click* action. The EnterData rule is similar, except

$$(1)\quad \frac{}{\Sigma \vdash \epsilon \rightsquigarrow \epsilon} \qquad (5)\quad \frac{}{\Sigma \vdash x \rightsquigarrow x}$$

$$(2)\quad \frac{}{\Sigma \vdash \varrho \rightsquigarrow \Sigma[\varrho]} \qquad (6)\quad \frac{}{\Sigma \vdash \vartheta \rightsquigarrow \Sigma[\vartheta]}$$

$$(3)\quad \frac{\Sigma \vdash n \rightsquigarrow \rho}{\Sigma \vdash n/\phi[i] \rightsquigarrow \rho/\phi[i]} \qquad (7)\quad \frac{\Sigma \vdash v \rightsquigarrow \theta}{\Sigma \vdash v[key] \rightsquigarrow \theta[key]}$$

$$(4)\quad \frac{\Sigma \vdash n \rightsquigarrow \rho}{\Sigma \vdash n/\!\!/\phi[i] \rightsquigarrow \rho/\!\!/\phi[i]} \qquad (8)\quad \frac{\Sigma \vdash v \rightsquigarrow \theta}{\Sigma \vdash v[i] \rightsquigarrow \theta[i]}$$

$$(9)\quad \frac{\Sigma \vdash n \rightsquigarrow \rho \quad \Sigma \vdash n \rightsquigarrow \rho}{\Sigma \vdash \mathit{Children}(n, \phi)_{\geq i} \rightsquigarrow \rho/\phi[i] :: \mathit{Children}(\rho, \phi)_{\geq i+1}}$$

$$(10)\quad \frac{\Sigma \vdash n \rightsquigarrow \rho}{\Sigma \vdash \mathit{Dscts}(n, \phi)_{\geq i} \rightsquigarrow \rho/\!\!/\phi[i] :: \mathit{Dscts}(\rho, \phi)_{\geq i+1}}$$

$$(11)\quad \frac{\Sigma \vdash v \rightsquigarrow \theta \quad arr = \mathit{GetArray}(\Sigma[x], \theta)}{\Sigma \vdash \mathit{ValuePaths}(v) \rightsquigarrow \left[\theta[1], \cdot\cdot, \theta[|arr|]\right]}$$

Figure 8. Auxiliary rules for our trace semantics.

that it also evaluates the value path expression v. As we can see, these rules form the base cases of our semantics.

Evaluating loopy statements. The rest of the rules from Figure 7 deal with loops. Amongst the first three rules that handle selector loops, the most interesting one perhaps is S-Cont: it unrolls the loop once if the first selector ρ refers to a DOM node that exists in π_1 (checked by *valid*). This is another example for how we use DOMs to guide the simulated execution: we use DOMs to handle branches in loops. If ρ exists in π_1 (e.g., the next element to be scraped exists), we bind ϱ to ρ and execute the loop body P. Note that S-Cont unrolls loops lazily. This is because many websites load more DOM nodes while scrolling down a page: we cannot eagerly fetch all DOM nodes at the beginning; instead, we have to keep executing until all nodes are loaded. The next rule, VP-Loop, handles value path loops. It is eager and it iterates over all value paths in V. The last three rules handle while loops. A key distinction here is the termination condition: while loops are *click-terminated*. That is, if the selector in the last *Click* is not valid, it terminates. As mentioned earlier, this is mainly used to handle pagination using "next page". Note that, though not explicitly being defined, we use a standard `if` construct in our rules to help formalize the semantics.

Auxiliary rules. Figure 8 presents the auxiliary rules for evaluating symbolic selectors and value paths. They are fairly straightforward. For instance, rules (1)-(4) handle selector expressions that may contain variables, by basically replacing variables with concrete values. Rules (5)-(8) are conceptually the same except that they are for symbolic value paths. Rules (9)-(11) evaluate selectors expressions.

Example 3.1. Consider the following program P, which is an extremely simplified version of P_1 from Figure 5(c).

$$\texttt{foreach}\ \varrho\ \texttt{in}\ \mathit{Dscts}(\epsilon, a)\ \texttt{do}\ \{\mathit{Click}(\varrho)\}$$

Here, P performs a *Click* (using variable ϱ) in a selectors loop. For simplicity, let us consider a DOM trace $\Pi = [\pi_1, \pi_2]$. Let us also assume Π indeed corresponds to P; that is, the i-th click in P executed on DOM π_i transitions the page to π_{i+1}.

$$\dfrac{\dfrac{\dfrac{\{x \mapsto \bot, \varrho \mapsto /\!/a[2]\} \vdash \varrho \rightsquigarrow /\!/a[2]}{[\pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\} \vdash Click(\varrho) \rightsquigarrow [Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{Click}) \quad \dfrac{}{[], \cdot\cdot \vdash \cdot\cdot \rightsquigarrow [], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{Term})}{[\pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\} \vdash Click(\varrho); \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)_{\geq 3}\ \texttt{do}\ \{Click(\varrho)\} \rightsquigarrow [Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{Seq})$$

$$\dfrac{\{\varrho \mapsto /\!/a[1]\} \vdash Dscts(\epsilon, a)_{\geq 2} \rightsquigarrow /\!/a[2] :: Dscts(\epsilon, a)_{\geq 3} \quad valid(/\!/a[2], \pi_2) \quad \boxed{\text{See above.}}}{[\pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[1]\} \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)_{\geq 2}\ \texttt{do}\ \{Click(\varrho)\} \rightsquigarrow [Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{S-Cont})$$

$$\dfrac{\dfrac{\dfrac{\{x \mapsto \bot, \varrho \mapsto /\!/a[1]\} \vdash \varrho \rightsquigarrow /\!/a[1]}{[\pi_1, \pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[1]\} \vdash Click(\varrho) \rightsquigarrow [Click(/\!/a[1])], [\pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[1]\}}(\textsc{Click}) \quad \boxed{\text{See above.}}}{[\pi_1, \pi_2], \{x \mapsto \bot, \varrho \mapsto /\!/a[1]\} \vdash Click(\varrho); \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)_{\geq 2}\ \texttt{do}\ \{Click(\varrho)\} \rightsquigarrow [Click(/\!/a[1]), Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{Seq})$$

$$\dfrac{\dfrac{\dfrac{\{x \mapsto \bot\} \vdash Dscts(\epsilon, a)_{\geq 1} \rightsquigarrow /\!/a[1] :: Dscts(\epsilon, a)_{\geq 2} \quad valid(/\!/a[1], \pi_1) \quad \boxed{\text{See above.}}}{[\pi_1, \pi_2], \{x \mapsto \bot\} \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)_{\geq 1}\ \texttt{do}\ \{Click(\varrho)\} \rightsquigarrow [Click(/\!/a[1]), Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{S-Cont})}{[\pi_1, \pi_2], \{x \mapsto \bot\} \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)\ \texttt{do}\ \{Click(\varrho)\} \rightsquigarrow [Click(/\!/a[1]), Click(/\!/a[2])], [], \{x \mapsto \bot, \varrho \mapsto /\!/a[2]\}}(\textsc{S-Init})}{[\pi_1, \pi_2], \bot \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)\ \texttt{do}\ \{Click(\varrho)\} : [Click(/\!/a[1]), Click(/\!/a[2])]}(\textsc{Eval})$$

Figure 9. A derivation for the program in Example 3.1 using our trace semantics.

We illustrate our semantics on P; Figure 9 shows its derivation. First of all, Eval returns two actions that are executed in the first two iterations of P, since Π has two DOMs. The *valid* checks in S-Cont are used to guide our simulated execution. For P, these checks all pass, as P indeed produced Π. However, consider the following P':

$$\texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)\ \texttt{do}\ \{Click(\varrho/b)\}$$

For P', the checks may not pass as "$/\!/a[1]/b$" might not refer to a valid node in DOM π_1. In that case, we will invoke the S-Term rule, eventually producing a shorter action trace.

4 Web RPA Program Synthesis Problem

In this section, we formulate our program synthesis problem.

Definition 4.1. (Satisfaction). Given input data I, an action trace A and a DOM trace Π, a web RPA program P *satisfies* A, if we have (1) $\Pi, I \vdash P : A'$ and (2) A is *consistent* with a *prefix* of A' with respect to Π. In other words, P can *reproduce* A.

Definition 4.1 requires checking consistency between two traces of actions. To do this, we first define two actions a_1 and a_2 to be consistent, given a DOM π, if a_1 and a_2 are of the same type and their arguments match. Note that two XPath arguments match each other, if they refer to the same DOM node on π. Then, two action traces A_1 and A_2 are consistent, given a DOM trace Π, if the i-th action in A_1 is consistent with the i-th action in A_2 given the i-th DOM in Π.

The reason that condition (2) uses "prefix", instead of requiring A to be consistent with A' exactly, is because A is in general an *incomplete* trace. That is, A may be a prefix of the entire action trace of P. In other words, our trace semantics might produce a longer action trace than the demonstration.

Definition 4.2. (Generalization). Given input data I, an action trace A and a DOM trace Π, a web RPA program P *generalizes* A, if we have (1) $\Pi, I \vdash P : A'$ and (2) A is consistent with a *strict* prefix of A' given Π. In other words, P not only reproduces A but also executes more actions after A.

Definition 4.2 requires "strict prefix", as our goal is to predict unseen actions beyond only reproducing those observed.

Definition 4.3. (Web RPA Program Synthesis Problem). Given input data I, an action trace $A = [a_1, \cdot\cdot, a_m]$ and a DOM trace $\Pi = [\pi_1, \cdot\cdot, \pi_{m+1}]$, find a web RPA program P that *generalizes* A, given I and Π.

Intuitively, Definition 4.3 takes as input A and Π where a_i is an action performed on π_i, and it looks for a program P that can be used to *predict* an action a_{m+1} that might be performed on π_{m+1}. In general, we require Π be longer than A; otherwise, we are not able find a program that generalizes. In practice, we require Π have one more element than A, because we can obtain the latest DOM without knowing the user's next action on it. Also note that, we may have multiple programs that generalize A; therefore, we aim to synthesize a *smallest* program in size.

5 Web RPA Program Synthesis Algorithm

5.1 Top-level rewrite-based synthesis algorithm

Algorithm 1 shows the top-level synthesis algorithm. Our key idea is to *iteratively rewrite* the action trace A into a program that generalizes A given DOM trace Π and input data I. The algorithm is not destructive and maintains intermediate rewrites; it heuristically picks a "best" program at the end.

Algorithm 1 maintains a worklist of tuples $(P, \vec{A}, \vec{\Pi})$, where $P = S_1; \cdot\cdot; S_l$ is a program rewritten from the input trace A. $\vec{A} = [A_1, \cdot\cdot, A_l]$ is a list of action traces, and $\vec{\Pi} = [\Pi_1, \cdot\cdot, \Pi_l]$ is a list of DOM traces. We maintain the following invariant:

$$\mathcal{I}_1: \quad A_1 +\!\!+ \cdot\cdot +\!\!+ A_l = A \text{ and } \Pi_1 +\!\!+ \cdot\cdot +\!\!+ \Pi_l = [\pi_1, \cdot\cdot, \pi_m]$$

Essentially, $\mathcal{I}_1$ says $\vec{A}$ is a partition of A and $\vec{\Pi}$ is a partition of the first m DOMs in Π. It is fairly easy to show $\mathcal{I}_1$ holds for $P_0, \vec{A}_0, \vec{\Pi}_0$. The second invariant is:

$$\mathcal{I}_2: \quad \forall i \in [1, l], S_i \textit{ satisfies } A_i \textit{ given } I \textit{ and } \Pi_i$$

which says that each S_i in P *satisfies* the corresponding *slice* A_i. This is also trivially true for $P_0, \vec{A}_0, \vec{\Pi}_0$, as every S_i in P_0 is a single loop-free statement. These invariants guarantee our

procedure SYNTHESIZE (A, Π, I)
input: $A = [a_1, \cdot\cdot, a_m]$, $\Pi = [\pi_1, \cdot\cdot, \pi_{m+1}]$, and input data I.
output: a program P that generalizes A given Π and I.
1: $P_0 := a_1; \cdot\cdot; a_m$; $\vec{A}_0 := \big[[a_1], \cdot\cdot, [a_m]\big]$; $\vec{\Pi}_0 := \big[[\pi_1], \cdot\cdot, [\pi_m]\big]$;
2: $W := \{(P_0, \vec{A}_0, \vec{\Pi}_0)\}$; $\widetilde{P} := \emptyset$;
3: **while** $W \neq \emptyset$
4: $(P, \vec{A}, \vec{\Pi}) := W$.remove();
5: **if** A' *generalizes* A *given* Π *and* Σ **then** $\widetilde{P}$.add(P);
6: $\Omega := \text{SPECULATE}(P, \vec{A}, \vec{\Pi})$;
7: $W' := \text{VALIDATE}(\Omega, P, \vec{A}, \vec{\Pi})$; $W := W \cup W'$;
8: **return** RANK($\widetilde{P}$);

Algorithm 1. Top-level synthesis algorithm.

rewrites always *satisfy* the specification A. Intuitively, this is because every statement S_i in P satisfies each slice A_i in A, thus the "concatenation" of all S_i's, that is P, would also satisfy the concatenation of all A_i's, which is A.

The worklist algorithm maintains $\mathcal{I}_1$ and $\mathcal{I}_2$, too. It tracks a worklist W of programs that *satisfy* A but only stores those *generalizable* programs into $\widetilde{P}$. In particular, the algorithm first removes a tuple $(P, \vec{A}, \vec{\Pi})$ from W (line 4). It then checks if P generalizes A; if so, it adds P into $\widetilde{P}$ (line 5). The algorithm grows the worklist using our *speculate-and-validate* method to rewrite P into more programs all of which maintain $\mathcal{I}_1$ and $\mathcal{I}_2$ (lines 6-7). Intuitively, given $P = S_1; \cdot\cdot; S_l$, this rewrite process replaces *a slice of statements* $S_i, \cdot\cdot, S_j$ in P with a loop statement S' such that $S_1; \cdot\cdot; S_{i-1}; S'; S_{j+1}; \cdot\cdot; S_l$ also meets $\mathcal{I}_1$ and $\mathcal{I}_2$. Note that, because a statement in P might itself be loopy, we can generate nested loops (from the inside out).

Challenges. While conceptually simple, this idea is technically quite challenging to realize. A key challenge is that, it is in general quite hard to encode all patterns as rules, if we follow standard rewrite-based synthesis approaches [42, 62]: our DSL has multiple types of loops, a loop body may have multiple statements that may use loop variables in different ways, loops can be nested, *etc.* There are too many cases. Even if we can define these rules, it is not clear how efficient this rule-based approach is, given the trace may correspond to an arbitrarily complex program. Let us further illustrate this challenge using the following example.

Example 5.1. Consider the following program P, which is a simplified version of P_2 in Figure 5(c). It scrapes information from a list of items spanned across multiple pages.

```
while true do
  foreach ϱ in Dscts(ϵ, a) do
    ScrapeText(ϱ)
    ScrapeText(ϱ/b)
  Click(c)
```

Suppose we are given the following action trace A for P:

$[\ \mathit{ScrapeText}(/\!/a[1]), \mathit{ScrapeText}(/\!/a[1]/b), \cdot\cdot,$
$\mathit{ScrapeText}(/\!/a[20]), \mathit{ScrapeText}(/\!/a[20]/b),$ $(a_1, \cdot\cdot, a_{40})$
$\mathit{Click}(c),$ (a_{41})
$\mathit{ScrapeText}(/\!/a[1]), \mathit{ScrapeText}(/\!/a[1]/b), \cdot\cdot,$
$\mathit{ScrapeText}(/\!/a[9]), \mathit{ScrapeText}(/\!/a[9]/b)\ ]$ $(a_{42}, \cdot\cdot, a_{59})$

procedure SPECULATE $(P, \vec{A}, \vec{\Pi})$
input: $P = S_1; \cdot\cdot; S_l$, $\vec{A} = [A_1, \cdot\cdot, A_l]$, $\vec{\Pi} = [\Pi_1, \cdot\cdot, \Pi_l]$.
output: a set Ω of speculative rewrites of the form (S', S_i, S_j).
1: $\Omega := \emptyset$;
2: **for** $i \le p \le j < q$ *s.t.* $[S_i, \cdot\cdot, S_p, \cdot\cdot, S_j, \cdot\cdot, S_q] \sqsubseteq [S_1, \cdot\cdot, S_l]$ *and* $j - i + 1 = q - p$
3: **for** $(S'_p, \varrho, N) \in \text{ANTI-UNIFY}(S_p, S_q)$
4: $\rho := \text{FirstSelector}(N)$;
5: $\widetilde{P'} := \big\{S'_i; \cdot\cdot; S'_p; \cdot\cdot; S'_j \,\big|\, S'_k \in \text{PARAMETRIZE}(S_k, \varrho, \rho), k \in [i, j] \setminus \{p\}\big\}$;
6: $\widetilde{S'} := \big\{\textbf{foreach}\ \varrho\ \textbf{in}\ N\ \textbf{do}\ P' \mid P' \in \widetilde{P'}\big\}$;
7: $\Omega := \Omega \cup \{(S', S_i, S_j) \mid S' \in \widetilde{S'}\}$;
8: **for** $i \le p \le j < q$ *s.t.* $[S_i, \cdot\cdot, S_p, \cdot\cdot, S_j, \cdot\cdot, S_q] \sqsubseteq [S_1, \cdot\cdot, S_l]$ *and* $j - i + 1 = q - p$
9: **for** $(S'_p, \vartheta, V) \in \text{ANTI-UNIFY}(S_p, S_q)$
10: $\theta := \text{FirstValuePath}(V)$;
11: $\widetilde{P'} := \big\{S'_i; \cdot\cdot; S'_p; \cdot\cdot; S'_j \,\big|\, S'_k \in \text{PARAMETRIZE}(S_l, \vartheta, \theta), k \in [i, j]\ \{p\}\big\}$;
12: $\widetilde{S'} := \big\{\textbf{foreach}\ \vartheta\ \textbf{in}\ V\ \textbf{do}\ P' \,\big|\, P' \in \widetilde{P'}\big\}$;
13: $\Omega := \Omega \cup \{(S', S_i, S_j) \mid S' \in \widetilde{S'}\}$;
14: **for** $i < p < q$ *s.t.* $[S_i, \cdot\cdot, S_p, \ldots, S_q] \sqsubseteq [S_1, \cdot\cdot, S_l]$ *and* $p - i + 1 = q - p$
15: **if** $S_p = S_q = \mathit{Click}(\rho)$ **then**
16: $S' := \textbf{while}\ \mathit{true}\ \textbf{do}\ \{S_i, \cdot\cdot, S_p\}$; $\Omega := \Omega \cup \{(S', S_i, S_j)\}$;
17: **return** Ω;

Algorithm 2. SPECULATE procedure.

Here, $a_1, \cdot\cdot, a_{41}$ correspond to *all* actions from the first iteration of the **while** loop, including 40 actions from **foreach**. The remaining actions $a_{42}, \cdot\cdot, a_{59}$ correspond to a partial execution of the second iteration of **while**. We also record a DOM trace $[\pi_1, \cdot\cdot, \pi_{60}]$ as well, where a_i is performed on π_i and π_{60} is the latest DOM.

In order to generate P from A, the standard rewrite-based synthesis approaches [42, 62] apply a set of predefined *sound* rewrite rules to rewrite A to P. Conceptually, it would use these rules to essentially identify iteration boundaries and repetitive patterns, in order to eventually "reroll" the trace back to the desired program with loops. That is an enormous space which might contain only a few correct rewrites.

In this work, we take a different route that incorporates the "guess-and-check" idea into the overall rewrite process. Our approach does not generate true rewrites *directly* using sound rewrite rules; instead, it first *speculates* likely rewrites which are then *validated* using our trace semantics.

5.2 Speculation

We first describe our speculation procedure; see Algorithm 2. It takes as input a program $P = S_1; \cdot\cdot; S_l$, a list of action traces $\vec{A} = [A_1, \cdot\cdot, A_l]$, and a list of DOM traces $\vec{\Pi} = [\Pi_1, \cdot\cdot, \Pi_l]$. SPECULATE returns a set Ω of *speculative rewrites*, or *s-rewrites*, of the form (S', S_i, S_j). Here, S' is a loop statement whose *first* iteration corresponds to $S_i; \cdot\cdot; S_j$ from P. That is, they yield the same trace (this is guaranteed by construction). However, an s-rewrite may not be a true rewrite: a true rewrite must have *more than one iterations* exhibited in P, but an s-rewrite is only guaranteed to have its first iteration exhibited in P. Nevertheless, s-rewrites have a very nice property: they are much easier to generate, and they over-approximate the set of true rewrites. Our technique makes use of this property.

To generate s-rewrites that *tightly over-approximate* the set of true rewrites, we follow existing rule-based approaches

$$(1)\quad \frac{\varrho \text{ fresh} \quad \varrho \vdash \rho_1 \circledast \rho_2 \twoheadrightarrow (n, N)}{\vdash Click(\rho_1) \circledast Click(\rho_2) \twoheadrightarrow (Click(n), \varrho, N)}$$

$$(2)\quad \frac{\begin{array}{c}\varrho' \text{ fresh} \quad \varrho' \vdash N_1 \circledast N_2 \twoheadrightarrow (N, N') \quad P_1, P_2 \text{ alpha equivalent} \\ U = (\texttt{foreach}\ \varrho_1\ \texttt{in}\ N\ \texttt{do}\ P_1, \varrho', N')\end{array}}{\vdash \texttt{foreach}\ \varrho_1\ \texttt{in}\ N_1\ \texttt{do}\ P_1 \circledast \texttt{foreach}\ \varrho_2\ \texttt{in}\ N_2\ \texttt{do}\ P_2 \twoheadrightarrow U}$$

$$(3)\quad \frac{\begin{array}{c}\vartheta \text{ fresh} \quad \theta_1 = \theta[1][o_1]\cdot\cdot[o_r] \quad \theta_2 = \theta[2][o_1]\cdot\cdot[o_r] \\ U = (EnterData(\rho, \vartheta[o_1]\cdot\cdot[o_r]), \vartheta, ValuePaths(\theta))\end{array}}{\vdash EnterData(\rho, \theta_1) \circledast EnterData(\rho, \theta_2) \twoheadrightarrow U}$$

$$(4)\quad \frac{\begin{array}{c}\exists \rho_1' \in AlternativeSelectors(\rho_1)\colon \rho_1' = n\left[\varrho \mapsto \rho/\phi[1]\right] \\ \exists \rho_2' \in AlternativeSelectors(\rho_2)\colon \rho_2' = n\left[\varrho \mapsto \rho/\phi[2]\right]\end{array}}{\varrho \vdash \rho_1 \circledast \rho_2 \twoheadrightarrow \left(n, Children(\rho, \phi)\right)}$$

$$(5)\quad \frac{\varrho \vdash \rho_1 \circledast \rho_2 \twoheadrightarrow (n, N)}{\varrho \vdash Children(\rho_1, \phi) \circledast Children(\rho_2, \phi) \twoheadrightarrow (Children(n, \phi), N)}$$

Figure 10. Anti-Unify rules.

in prior work [42, 62]. However, our rules are designed to detect patterns *partially*, instead of completely. A key step in our approach is to inspect two statements S_p, S_q in P and generate a loop S' *such that*, S_p, S_q correspond to the *same* statement from S' but S_p "comes" from its first iteration and S_q from its second. For example, lines 2-7 in Algorithm 2 generate selector loops. It first enumerates all slices $S_i, \cdot\cdot, S_j$ in P assuming S_i and S_j correspond to the start and end of the first iteration (line 2). Then, it tries to "merge" S_p, S_q into a *parametrized* statement S_p' by calling Anti-Unify (line 3). Similarly, lines 8-13 handle value path loops.

Anti-unification. In the context of logic programming, anti-unification [13, 14] refers to the process of generating two terms t_1 and t_2 into a least general template τ for which there exists substitutions α_1 and α_2, such that $\tau(\alpha_1) = t_1$ and $\tau(\alpha_2) = t_2$. It has been used in prior work [55] to generate code fixes; in this work, we use anti-unification to synthesize loops. Figure 10 gives some representative rules. In a nutshell, our procedure returns a set of tuples (S_p', ϱ, N), where S_p' is a more general statement using loop variable ϱ in the target loop S', and N is the selectors that S' loops over.

Let us take rule (1) as an example. Here, it anti-unifies two *Click* statements whose selectors differ at only one index in their XPath expressions. Specifically, it calls rule (4) that anti-unifies selectors ρ_1 and ρ_2 given a fresh variable ϱ. Intuitively, it looks for a general selector n that uses variable ϱ such that n instantiates to ρ_1' and ρ_2', respectively. Note that rule (4) considers alternative selectors; this is necessary for inducing more general programs. Rule (4) also returns $Children(\rho, \phi)$ which is the collection that the target loop statement S' loops over. We have a very similar rule that anti-unifies ρ_1, ρ_2 and generates $Dscts(\rho, \phi)$, though it is not shown here.

Rule (2) anti-unifies two selector loops by anti-unifying their respective collections N_1 and N_2, which is conditional on their loop bodies P_1, P_2 being alpha-equivalent. Rule (3) performs anti-unification for *EnterData* statements.

Example 5.2. Consider the action trace A from Example 5.1. Line 2 of our Speculate procedure will consider all possible tuples (i, p, j, q), where i, j are the start and end of the first iteration of a loop to be generated. Consider $(1, 1, 2, 3)$, which corresponds to the first iteration of the `foreach` loop in P. Anti-Unify at line 3 generates $(ScrapeText(\varrho), \varrho, Dscts(\epsilon, a))$ from S_p, S_q, which are a_1, a_3 in this case. Here, $ScrapeText(\varrho)$ is the desired statement in the `foreach` loop's body. Furthermore, it also gives the selectors expression, $Dscts(\epsilon, a)$, that the target `foreach` loop iterates over. Yet, our Anti-Unify procedure does not generate the rest of the body.

$$(1)\ \frac{S = Click(\rho)}{\varrho, \rho' \vdash Click(\rho) \twoheadrightarrow S} \quad (2)\ \frac{\exists \rho'' \in AlternativeSelectors(\rho)\colon \rho'' = \rho'/\rho'''}{\varrho, \rho' \vdash Click(\rho) \twoheadrightarrow Click(\varrho/\rho''')}$$

$$(3)\ \frac{S = \texttt{foreach}\ \varrho\ \texttt{in}\ N\ \texttt{do}\ P}{\varrho', \rho \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N\ \texttt{do}\ P \twoheadrightarrow S}$$

$$(4)\ \frac{\varrho', \rho \vdash N \twoheadrightarrow N'}{\varrho', \rho \vdash \texttt{foreach}\ \varrho\ \texttt{in}\ N\ \texttt{do}\ P \twoheadrightarrow \texttt{foreach}\ \varrho\ \texttt{in}\ N'\ \texttt{do}\ P}$$

$$(5)\ \frac{N = Children(\rho, \phi)}{\varrho, \rho' \vdash Children(\rho, \phi) \twoheadrightarrow N} \quad (6)\ \frac{\rho = \rho'/\rho'' \quad N = Children(\varrho/\rho'', \phi)}{\varrho, \rho' \vdash Children(\rho, \phi) \twoheadrightarrow N}$$

Figure 11. Parametrize rules.

Parametrization Anti-Unify essentially creates a skeleton of the entire loop S': it gives one statement in the loop body but we still need to construct the rest. This is exactly what Algorithm 2 does at lines 4-7. In particular, it first obtains the binding $\varrho \mapsto \rho$ in the first iteration. Then, given this binding, it uses the Parametrize procedure to construct the entire loop. Figure 11 presents some representative rules. For instance, rules (1) and (2) parametrize a *Click* statement. Rule (1) keeps the *Click* as is, since it is possible that a statement inside a loop does not use the variable. Rule (2) parametrizes the *Click* if the selector ρ' that variable ϱ binds to is a prefix of some *alternative selector* for the argument ρ in *Click*. Rules (3) and (4) parametrize a selectors loop in a very similar way, though it uses additional rules (5) and (6) to handle selectors.

Example 5.3. Consider the output of Anti-Unify, namely, $(ScrapeText(\varrho), \varrho, Dscts(\epsilon, a))$, in Example 5.2. Given this output, line 4 of Algorithm 2 obtains the first selector ρ of $Dscts(\epsilon, a)$; that is, $\rho = /\!/a[1]$. Then, we parametrize each of the remaining statements within $[i, j]$—in our case, only a_2. One statement given by Parametrize is $ScrapeText(\varrho/b)$, which is the desired statement in P. Therefore, $\widetilde{S'}$ at line 6 of Algorithm 2 includes the desired `foreach` loop. Finally, line 7 adds the following s-rewrite to Ω.

$$\left(\begin{array}{l}\texttt{foreach}\ \varrho\ \texttt{in}\ Dscts(\epsilon, a)\ \texttt{do} \\ \quad ScrapeText(\varrho) \\ \quad ScrapeText(\varrho/b)\end{array}, \ a_1, \ a_2\right)$$

While this loop corresponds to $a_1, \cdot\cdot, a_{40}$, our Speculate procedure only guarantees its first iteration corresponds to a_1, a_2.

5.3 Validation

As we can see, s-rewrites are fairly easy to generate but may be spurious. That is, they might not rewrite beyond the first iteration. Can we filter them out, and if so, how? Our idea is to validate them using our trace semantics; see Algorithm 3. In

procedure VALIDATE ($\Omega, P, \vec{A}, \vec{\Pi}$)
input: a set Ω of s-rewrites of the form (S', S_i, S_j).
input: a program $P = S_1; \cdot\cdot; S_l$ that each s-rewrite in Ω may apply to.
input: $\vec{A} = [A_1, \cdot\cdot, A_l]$, $\vec{\Pi} = [\Pi_1, \cdot\cdot, \Pi_l]$.
output: a set W of true rewrites of the form $(P', \vec{A}', \vec{\Pi}')$ *s.t.* $\mathcal{I}_1$, $\mathcal{I}_2$ hold.
1: $W := \emptyset$;
2: **for** $(S', S_i, S_j) \in \Omega$
3: $A' :=$ EXECUTE$(S', \Pi_i{+}{+}\cdot\cdot{+}{+}\Pi_l, I)$;
4: **if** $\exists r \in [j+1, l] : A' = A_i{+}{+}\cdot\cdot{+}{+}A_r$ **then**
5: $P' := S_1; \cdot\cdot; S_{i-1}; S'; S_{r+1}; \cdot\cdot; S_l$;
6: $\vec{A}' := [A_1, \cdot\cdot, A_{i-1}, A', A_{r+1}, \cdot\cdot, A_l]$;
7: $\vec{\Pi}' := [\Pi_1, \cdot\cdot, \Pi_{i-1}, \Pi', \Pi_{r+1}, \cdot\cdot, \Pi_l]$;
8: $W := W \cup \{(P', \vec{A}', \vec{\Pi}')\}$;
9: **return** W;

Algorithm 3. VALIDATE procedure.

a nutshell, given an s-rewrite S' corresponding to $S_i, \cdot\cdot, S_j$, the algorithm checks whether it is a true rewrite or not; if so, it returns a slice of statements $S_i, \cdot\cdot, S_r$ in P that can be rewritten to S'. We require $r > j$, because we want S' to rewrite a slice of statements beyond S_j (i.e., the first iteration). Towards this goal, VALIDATE first executes S' against the concatenation of DOM traces from i to l, yielding an action trace A' (line 3). Then, line 4 checks if S' is a true rewrite; if so, it obtains the rewrite P' (line 5) and the matching traces (lines 6-7), which are then added to W (line 8). Note that invariants $\mathcal{I}_1, \mathcal{I}_2$ hold for this rewrite $(P', \vec{A}', \vec{\Pi}')$, as A' is obtained by executing S' using our trace semantics and is also checked at line 4.

Example 5.4. Consider the s-rewrite (S', a_1, a_2) returned by SPECULATE in Example 5.3. By construction, the first iteration of S' produces $[a_1, a_2]$. To validate this s-rewrite, we evaluate S' against $[\pi_1, \cdot\cdot, \pi_{60}]$ using our trace semantics, which gives an action trace $A' = [a_1, \cdot\cdot, a_{40}]$. This is indeed a true rewrite; thus, VALIDATE returns S' together with its matching action trace $[a_1, \cdot\cdot, a_{40}]$, indicating S' rewrites actions $a_1, \cdot\cdot, a_{40}$.

5.4 Incremental synthesis

Recall from Figure 3 that our synthesizer is used in an iterative fashion: it predicts the next action given the current trace with m actions, where m increases as the task progresses. Therefore, we invoke our synthesis algorithm *incrementally*. This is done by simply sharing the worklist in Algorithm 1 across synthesis runs. Suppose we want to synthesize from a trace with m actions, given the worklist W from the previous run. Instead of starting from scratch (line 2, Algorithm 1), we resume from $W \cup W'$, where W' contains those programs removed from W (line 4) in the previous run. This essentially makes the entire rewrite process across runs not destructive.

5.5 Soundness and completeness

Theorem 5.5. Given action trace A, DOM trace Π and input data I, if there exists a web RPA program that generalizes A and in which every loop has at least two iterations exhibited in A, then our synthesis algorithm would return a program that generalizes A given Π and I.

6 Human-in-the-loop Interaction Model

In this section, we describe our system interaction design rationale and user interface. Our overall design goal is to reduce the gulfs of execution and evaluation for novice users [44]. That is, through our interface, we aim to help users better understand what is going on in the system (i.e., evaluation) as well as help them execute intended actions (i.e., execution). To achieve this goal, we designed a user interaction model that combines PBD and user interaction in a human-in-the-loop process. We highlight some key features below.

Demo-auth-auto workflow. As illustrated in Section 2, there are three phases when using our tool: (a) a *demonstration* phase where the user manually performs a few actions, (b) an *authorization* phase where the user accepts or rejects predictions, and (c) an *automation* phase where our tool automatically executes the program. Our system could transition from one phase to another (automatically or manually).

Data entry via drag-and-drop. Instead of manually typing strings from the input data source, our interface supports drag-and-drop. This design not only simplifies the data entry process but also makes synthesis easier.

Action highlighting. Each action performed on the page is highlighted. In addition, during the demonstration phase, our system also highlights DOM nodes that are hovered over. These designs help users better interact with our system.

Prediction authorization. During the authorization phase, each predicted action requires user approval before it is executed. Our user interface visualizes predictions in an easy-to-examine manner, which helps reduce the gulf of evaluation.

Navigating across multiple predictions. In case there are multiple predictions, our interface will show a navigation arrow which allows users to inspect each of them and accept the desired one, which helps resolve ambiguity.

7 Evaluation

In this section, we describe a series of experiments that are designed to answer the following research questions:

- **Q1:** Can WEBROBOT's synthesis engine effectively synthesize web RPA programs from demonstrations?
- **Q2:** How important are the ideas proposed in Section 5?
- **Q3:** How well does WEBROBOT work *end-to-end* (including both front-end and back-end) in practice?
- **Q4:** How does WEBROBOT's performance compare against existing rewrite-based synthesis approaches?

Implementation. WEBROBOT was implemented with the proposed ideas as well as several additional optimizations.

Benchmarks. We also constructed a suite of benchmarks for web RPA. In particular, we first scraped all posts under the *"Data Extraction and Web Screen Scraping"* topic from the iMacros forum. Then, we retained every post that corresponds to a web RPA task with a working URL (e.g., we filter out posts regarding "how to use iMacros").

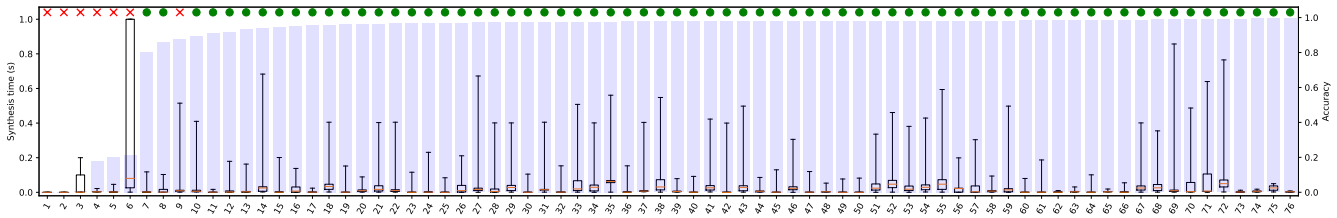

Figure 12. Main results for Q1: (a) accuracy (bar chart), (b) synthesis time (box plot), and (c) whether the final synthesized programs are intended (•/× marks near the top). In particular, benchmarks are sorted in ascending order based on accuracy. For each benchmark, we report quartile statistics of the synthesis times across all tests for which we can produce a prediction within the timeout (1 sec). If the final synthesized program is intended, we mark • at the top for that benchmark; otherwise, ×.

Ground-truth programs. For each benchmark, we have also manually written a program that can automate the corresponding task using the Selenium WebDriver framework. These programs are treated as the "ground-truth" programs.

Statistics of benchmarks. In total, we collected 76 benchmarks with their corresponding ground-truth programs. In particular, all of these benchmarks involve data extraction, 29 of them involve data entry, 60 require navigation across webpages, and 33 involve pagination. Some benchmarks may involve multiple types of actions: for instance, 28 of them involve data entry, data extraction, and webpage navigation. The ground-truth programs consist of 36.3 lines of code on average (max being 142). In general, it took us 30 minutes to a few hours to implement a working Selenium program.

7.1 Q1: Evaluating WebRobot's synthesis engine

Recall that our synthesizer takes as input (1) a demonstrated action trace $A = [a_1, \cdot\cdot, a_m]$, (2) a DOM trace $\Pi = [\pi_1, \cdot\cdot, \pi_{m+1}]$, and (3) an optional data source I. It returns a program P that generalizes A given Π and I. That is, P not only reproduces A but also predicts a next action a_{m+1}. Thus, our synthesis goal is to generate a_{m+1} efficiently and accurately.

Setup. To evaluate our synthesis efficiency and accuracy, we designed the following experiment. First, for each benchmark, we instrumented its ground-truth program P_{gt} such that P_{gt} would record every action it executes as well as all intermediate DOMs. Hence, we can obtain the *entire* action trace $A_{gt} = [a_1, \cdot\cdot, a_n]$ and DOM trace $\Pi_{gt} = [\pi_1, \cdot\cdot, \pi_n]$[7]. Here, a_1 is the first action performed on π_1 and a_n is the last action on π_n. We also ensure that the recorded actions are in the same trace language defined in Section 3. We convert the recorded selectors used in P_{gt} to *absolute* XPath expressions. The reason is because WebRobot's front-end records actions using absolute XPath during user interactions and we aim to simulate that in this experiment. Note that this actually makes synthesis more challenging since we necessarily need to consider alternative selectors in order to synthesize P_{gt}. For those benchmarks involving programmatic data entry, we manually constructed a data source I with 100 entries.[8]

Given A_{gt} and Π_{gt}, we generate $n-1$ tests for the synthesis engine. That is, for the kth test, we are given $A_k = [a_1, \cdot\cdot, a_k]$ with the first k actions in A_{gt} and $\Pi_{k+1} = [\pi_1, \cdot\cdot, \pi_{k+1}]$ with the first $k+1$ DOMs in Π_{gt}, and our goal is to synthesize a program that predicts a_{k+1}. In this setting, we define accuracy as the percentage of tests for which we can generate a *correct* prediction that is equivalent to the ground-truth action. For efficiency, we calculate the quartile statistics of the synthesis times across all tests that we can generate predictions for. In this experiment, we use 1 second as the timeout per test.

Main results. Overall, as shown in Figure 12, our synthesis engine solved most benchmarks with both high accuracy and efficiency. In particular, for 68% of the benchmarks, it achieves at least 95% accuracy within 0.5 seconds per prediction. Furthermore, it generates desired programs for 91% of the benchmarks. We note that it does not need the entire trace (with 500 actions) to generate those desired programs; rather, it typically generalizes with a few dozens of actions (and at most a couple hundreds). Also note that only a very small number of these actions (typically around 10) are *manually* demonstrated. Therefore, we believe our synthesis engine can be used in practice to interactively automate web RPA tasks. On average, the final synthesized programs have 6 statements and the largest program has 18. WebRobot can also synthesize programs with complex nesting structures: 32 of them involve doubly-nested loops and 6 involve at least three levels of nesting. Thus, we believe our synthesis engine has the potential to scale to complex web RPA tasks.

In what follows, we discuss some interesting findings.

Ambiguity. The synthesis engine generated multiple programs for 59 of our benchmarks. For 21 of them, it generated multiple predictions. The maximum numbers of synthesized programs and predictions are 101 and 6 respectively. This shows that web RPA is a domain with a fair amount of ambiguity, where there could exist multiple semantically different programs satisfying the same specification.

Pagination beyond "next page". Some websites use other mechanisms for pagination. For instance, *b9* involves a job search site[9] which performs pagination using page numbers and a "next 10 pages" button. We do not support such pagination mechanisms yet. The reason *b9* has an 88% accuracy is

[7] We terminate P_{gt} after 500 actions in case it unnecessarily takes long to finish. That is, we may use a prefix of P_{gt}'s entire trace in this experiment.
[8] Fun fact: we leveraged WebRobot when collecting these data sources.
[9] https://www.timesjobs.com/

because it synthesized a program with a sequence of selector loops, which solved the tests but is not intended.

Complex selectors. Some benchmarks need selectors with multiple attributes in order to be automated. For example, *b6* involves scraping players information for matches that have *either "match" or "match highlight" class*. Our DSL currently does not support such "disjunctive logics" for selectors. Some other benchmarks (such as *b1-3*) also have similar issues.

Others. The reason that *b7* has a relatively low accuracy (80%), albeit an intended program was synthesized, is because its trace is relatively short (with 51 actions in total) and the intended program was synthesized after the first 10 actions. This is also the case for some benchmarks, such as *b8*, *b10-12*.

7.2 Q2: Ablation studies of the synthesis engine

Setup. We performed ablation studies to quantify the impact of our ideas. In particular, we consider the following variants.

- *No selector*: We modified the *AlternativeSelectors* function from Figures 10 and 11 to always return the input selector. Effectively, this variant can only use full XPath expressions from the trace without considering alternative selectors.
- *No incremental*: This variant does not reuse rewrites from prior synthesis runs. It always starts from scratch if the program P_k generated for the kth test fails to predict a_{k+1}.

We conducted the same experiment described in Section 7.1 using these variants. Note that we do not include an ablation for the idea of speculative rewriting. The reason is because it is not easy to "disable" speculation without fundamentally changing our algorithm. Instead, please see Section 7.4 for a comparison against a baseline implemented using egg.

Main results. As shown in Table 1, our main take-away is that it is important to perform selector search and incremental synthesis in order to synthesize programs both accurately and efficiently. For instance, without considering alternative selectors, it only solves 38 benchmarks and the average accuracy drops to 57%. In terms of synthesis time, all techniques are fairly efficient (for tests that terminate within 1 second).

7.3 Q3: Evaluating WebRobot end-to-end

User study setup. To evaluate WebRobot end-to-end and access whether it helps users complete web RPA tasks, we conducted a user study involving 8 participants (avg. age 21) from the lead author's institution. All participants are undergraduate students with an average of 4 years of programming experience. Each participant was asked to complete 5 tasks sampled from our benchmarks, divided in three phases.

- *Phase 1* has one single-page scraping task.
- *Phase 2* includes two scraping tasks that involve webpage navigation and pagination.
- *Phase 3* has two tasks that involve data entry. In particular, given a list of keywords, the user needs to perform search on the website using each keyword and then scrape certain information from the search result of each keyword.

Variants	# Benchmarks solved	Accuracy (median)	Accuracy (average)	Time per test (average)
Full-fledged	69	98%	90%	23ms
No selector	38	88%	57%	54ms
No incremental	45	96%	72%	32ms

Table 1. Main results for ablation studies in Q2.

For each phase, participants started by watching a tutorial and replicating a demo task. Then, they worked on the main tasks. Each participant had one hour in total for all 5 tasks.

User study results. All participants were able to successfully automate all tasks using WebRobot. In particular, participants demonstrated 6-10 actions before WebRobot could automate the rest of the task. For each phase, the average time it took them to provide demonstrations is (in seconds): 16.88 (SD=3.80, phase 1), 19.44 (SD=11.48, phase 2), and 64.44 (SD=22.58, phase 3). Furthermore, all participants were able to use WebRobot to resolve ambiguity interactively. Finally, according to a follow-up survey, all eight participants gave positive feedback on the usability of our tool: for instance, they thought WebRobot was "quite decent" (*P4*), experience was "smooth" (*P8*), and it "can apply to many scenarios" (*P2*).

More comprehensive end-to-end testing results. To gain a more comprehensive understanding on how WebRobot works end-to-end, we tested it on *all* of our 76 benchmarks. While this experiment is inevitably biased because the testers are developers of the tool, we believe it complements the user study and hence is still valuable. A benchmark is counted as "solved" if we can use WebRobot to synthesize the intended program which can automate the benchmark.[10] Overall, we solved 76% of the benchmarks by interactively demonstrating around 10 actions. We also found it necessary to resolve ambiguity when solving these benchmarks. Among the remaining 18 benchmarks, 7 failed due to the issues from WebRobot's back-end (see Section 7.1), and 11 were due to limitations of our front-end. For instance, our current front-end does not fully support replaying certain actions in a few situations, which accounts for 7 cases.

Discussion. Comparing the end-to-end testing conducted by ourselves (i.e., WebRobot developers) and the user study with novice users, a notable difference in use patterns is that novice users make mistakes (e.g., mis-clicks on webpages and mis-use of the UI). In this case, we assisted the participants to restart WebRobot and perform the task again.

7.4 Q4: Comparison with existing rewrite-based synthesis techniques

The goal of this final experiment is to understand how our speculative rewriting idea compares with existing rule-based rewrite approaches that perform synthesis in a correct-by-construction manner (without speculation). Specifically, we implemented a baseline synthesizer using the egg library [62], a state-of-the-art framework that was used to build many high-performance rewrite-based synthesizers [42, 47, 57, 63].

[10] For long-running programs (>10 minutes), we ran them for 3 iterations.

	b12	*b15*	*b20*	*b48*	*b56*	*b73*	*b74*	*b75*	*b76*
Baseline using egg	$2{\times}10^5$/34	12/6	15/12	6/8	−/−	2/2	2/2	3/2	2/2
WebRobot	186/34	11/6	22/12	12/8	950/204	7/2	6/2	7/2	6/2

Table 2. Main results for egg-based implementation in Q4. X/Y gives synthesis time X (milliseconds) for (shortest) trace length Y.

Our* egg*-based implementation. Our baseline consists of two key rules: one that splits a trace into slices and another that "rerolls" a slice into a loop. We illustrate these rules using Example 5.1, by showing one specific sequence of rules that rewrites A to P. Conceptually, we first apply a *Split* rule that splits A to three slices:

$$[a_1, \cdot\cdot, a_{59}] \rightarrow \mathit{Unsplit}([[a_1, \cdot\cdot, a_{40}], [a_{41}], [a_{42}, \cdot\cdot, a_{59}]])$$

Then, we use a *Reroll* rule that yields two rewrites:

$$[a_1, \cdot\cdot, a_{40}] \rightarrow \mathit{InnerLoop} \quad [a_{42}, \cdot\cdot, a_{59}] \rightarrow \mathit{InnerLoop}$$

In other words, these two slices are rewritten to two instances of the inner loops of P. Note that now the e-graph contains $\mathit{Unsplit}([[\mathit{InnerLoop}], [a_{41}], [\mathit{InnerLoop}]])$. The third rule we apply is *Unsplit* which does the following rewrite:

$$\mathit{Unsplit}([[\mathit{InnerLoop}], [a_{41}], [\mathit{InnerLoop}]]) \rightarrow [\mathit{InnerLoop}, a_{41}, \mathit{InnerLoop}]$$

As one can imagine, we can apply the aforementioned rules again to finally generate P. While our current baseline only supports selector loops without alternative selectors, it leverages e-class analysis and is fairly optimized. Thus, we believe it is still a good baseline to test the performance of a purely rule-based, correct-by-construct synthesis approach.

Results and discussion. We evaluated this baseline on *all* 9 benchmarks whose ground-truth programs involve only selector loops and no alternative selectors. In particular, we ran it on action traces of increasing length (from length 1). Table 2 presents the synthesis time for the shortest trace, for which it gives an intended program, across *all* benchmarks. Our main take-away is that the baseline does not scale well. In particular, it solved 7 tasks whose ground-truth programs all have one single loop. *b12*, which requires synthesizing a doubly-nested loop, took 200s. The most complex problem is *b56*, which involves a three-level loop, and it did not terminate in 5 minutes. On the other hand, WebRobot solved all 9 benchmarks with at most 1 second.

8 Related Work

In this section, we briefly discuss some closely related work.

RPA. As a relatively new topic, there is little work on RPA until very recently [9, 33, 34, 61, 64]. Existing work mainly focuses on formalizing key concepts and the routine discovery problem. In contrast, our work targets a different problem of how to help non-experts create automation programs, which is also critical for fostering RPA adoption.

Web automation. Similar to web automation, WebRobot emulates user interactions with a web browser and hence can be viewed as a form of web automation. Our work differs from prior web automation work [1, 2, 4, 11, 16–18, 24, 35, 37, 38] in several ways. One notable difference is that WebRobot is based on interactive PBD whereas prior techniques are either program-centric or programmer-centric.

Interactive program synthesis. Different from prior approaches [12, 22, 26, 28, 32, 59], which are mostly interactive *programming-by-example*, WebRobot is based on interactive *programming-by-demonstration* which is a natural approach for web RPA. While action traces can be viewed as a form of "examples", it introduces a new challenge in how to define the correctness of a program against a trace. We use a form of trace semantics for our language, which lays the formal foundation for web RPA program synthesis.

Programming-by-demonstration (PBD). Existing PBD techniques can be roughly categorized into two groups: those that reason about *user actions* (e.g., Helena [18] and TELS [40]) and those that examine *application states* (e.g., Tinker [36] and SMARTedit [29]). While almost all of them rely on heuristic rules to generalize programs from demonstrations [30], a notable exception is SMARTedit, which proposes a principled approach based on version space algebras that could generate programs from a short trace of *states*. Similarly, our work contributes a principled approach but for *action-based trace generalization*. In particular, we propose a rewrite-based algorithm for synthesizing programs from a trace of actions.

Term rewriting. Term rewriting [21] has been used widely in many important domains [15, 20, 27, 42, 47, 48, 53, 56, 58, 62]. Our work explores a new application for synthesizing web RPA programs from traces. Different from existing rewrite-based synthesis techniques [42, 62] which are mostly purely rule-based and correct-by-construction, we leverage the idea of guess-and-check in the overall rewrite process. This new speculate-and-validate methodology enables and accelerates web RPA program synthesis.

Program synthesis with loop structures. WebRobot is related to a line of work that aims to synthesize programs with *explicit* loop structures [18, 22, 46, 52]. The key distinction is that we use demonstrations as specifications, whereas prior approaches are mostly programming-by-example.

Human-in-the-loop. WebRobot adopts a human-in-the-loop interaction model which has shown to be an effective way to incorporate human inputs when training AI systems in the HCI community [19, 45]. This model has been used in the context of program synthesis, mostly interactive PBE [23, 41, 43, 51, 60, 65]. In contrast, our work incorporates human inputs in PBD and proposes a new human-in-the-loop model.

Acknowledgments

We thank our shepherd, Calvin Smith, the PLDI'22 anonymous reviewers, Yuepeng Wang, Chenglong Wang, and Cyrus Omar for their valuable feedback. We also would like to thank Yiliang Liang and Minhao Li for their help on implementing our egg baseline. This work was supported by the National Science Foundation under Grant No. 2123654.

References

[1] Cypress Studio. https://docs.cypress.io/guides/core-concepts/cypress-studio.

[2] iMacros. https://www.progress.com/imacros.

[3] Robotic Process Automation (RPA). https://searchcio.techtarget.com/definition/RPA.

[4] Selenium IDE. https://www.selenium.dev/selenium-ide/.

[5] The Remarkable History of Robotic Process Automation (RPA). https://nandan.info/history-of-robotic-process-automation-rpa/.

[6] UiPath Webinar. https://www.uipath.com/webinar-recording/your-own-idea-robot-studiox?mkt_tok=OTk1LVhMVC04ODYAAAF8uBLrLqPW-QJHu_Hj1dkXeqK4JMZymY9EGBLkwL_2fSN8Kj2iwc09MVhHrBjf7PUkFUKBfYX-x-85mrFVUXZf2LawwpNcRPLTEDaZ9NM1.

[7] UiPath Webinar Slides. https://start.uipath.com/rs/995-XLT-886/images/StudioX_Webinar.pdf.

[8] XPath. https://en.wikipedia.org/wiki/XPath.

[9] Simone Agostinelli, Andrea Marrella, and Massimo Mecella. 2020. Towards Intelligent Robotic Process Automation for BPMers. *arXiv preprint arXiv:2001.00804* (2020).

[10] Tobias Anton. 2005. XPath-Wrapper Induction by Generalizing Tree Traversal Patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*. 126–133.

[11] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 748–764.

[12] Daniel W Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. *ACM SIGPLAN Notices* 50, 6 (2015), 218–228.

[13] Alexander Baumgartner and Temur Kutsia. 2014. Unranked second-order anti-unification. In *International Workshop on Logic, Language, Information, and Computation*. Springer, 66–80.

[14] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. 2017. Higher-order pattern anti-unification in linear time. *Journal of Automated Reasoning* 58, 2 (2017), 293–310.

[15] James M Boyle, Terence J Harmer, and Victor L Winter. 1997. The TAMPR program transformation system: Simplifying the development of numerical software. In *Modern software tools for scientific computing*. Springer, 353–372.

[16] 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.

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

[18] 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.

[19] Yan Chen, Jaylin Herskovitz, Walter S Lasecki, and Steve Oney. 2020. Bashon: A Hybrid Crowd-Machine Workflow for Shell Command Synthesis. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–8.

[20] Miles Claver, Jordan Schmerge, Jackson Garner, Jake Vossen, and Jedidiah McClurg. 2021. ReGiS: Regular Expression Simplification via Rewrite-Guided Synthesis. *arXiv preprint arXiv:2104.12039* (2021).

[21] Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite systems. In *Formal models and semantics*. Elsevier, 243–320.

[22] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: interactive program synthesis with control structures. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[23] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 614–626.

[24] Michael H Fischer, Giovanni Campagna, Euirim Choi, and Monica S Lam. 2021. DIY Assistant: A Multi-Modal End-User Programmable Virtual Assistant. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 312–327.

[25] Pankaj Gulhane, Amit Madaan, Rupesh Mehta, Jeyashankher Ramamirtham, Rajeev Rastogi, Sandeep Satpal, Srinivasan H Sengamedu, Ashwin Tengli, and Charu Tiwari. 2011. Web-scale information extraction with vertex. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1209–1220.

[26] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[27] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314.

[28] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372.

[29] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156.

[30] Tessa Ann Lau. 2001. *Programming by demonstration: a machine learning approach*. University of Washington.

[31] Tessa A Lau and Daniel S Weld. 1998. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the 4th international conference on Intelligent user interfaces*. 145–152.

[32] 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.

[33] Volodymyr Leno, Adriano Augusto, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, and Artem Polyvyanyy. 2021. Discovering Executable Routine Specifications from User Interaction Logs. *arXiv preprint arXiv:2106.13446* (2021).

[34] Volodymyr Leno, Stanislav Deviatykh, Artem Polyvyanyy, Marcello La Rosa, Marlon Dumas, and Fabrizio Maria Maggi. 2020. Robidium: Automated Synthesis of Robotic Process Automation Scripts from UI Logs. CEUR Workshop Proceedings.

[35] 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.

[36] Henry Lieberman. 1993. Tinker: A programming by demonstration system for beginning programmers. In *Watch what I do: programming by demonstration*. 49–64.

[37] 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.

[38] 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.

[39] Toshiyuki Masui and Ken Nakayama. 1994. Repeat and Predict - Two Keys to Efficient Text Editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 118–130.

[40] Dan Hua Mo. 1990. Learning Text Editing Procedures from Examples. (1990).

[41] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code using Query-by-Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 84–99.

[42] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–44.

[43] Julie L Newcomb and Rastislav Bodik. 2019. Using human-in-the-loop synthesis to author functional reactive programs. *arXiv preprint arXiv:1909.11206* (2019).

[44] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[45] Besmira Nushi, Ece Kamar, Eric Horvitz, and Donald Kossmann. 2017. On human intellect and machine failures: Troubleshooting integrative machine learning systems. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[46] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing data structure refinements from integrity constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 574–587.

[47] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices* 50, 6 (2015), 1–11.

[48] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1066–1082.

[49] Saikat Ray, Arthur Villa, Naved Rashid, Paul Vincent, Keith Guttridge, and Melanie Alexander. 2021. Magic Quadrant for Robotic Process Automation. https://www.gartner.com/doc/reprints?id=1-26Q65VFT&ct=210706&st=sb.

[50] Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1967–1978.

[51] Mark Santolucito, William T Hallahan, and Ruzica Piskac. 2019. Live programming by example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–4.

[52] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. Frangel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[53] Calvin Smith and Aws Albarghouthi. 2019. Program synthesis with equivalence reduction. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 24–47.

[54] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

[55] Reudismam Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Brazilian Symposium on Software Engineering*. 74–83.

[56] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[57] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. [n.d.]. Vectorization for Digital Signal Processors via Equality Saturation Extended Abstract. ([n. d.]).

[58] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building program optimizers with rewriting strategies. *ACM Sigplan Notices* 34, 1 (1998), 13–26.

[59] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

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

[61] Judith Wewerka and Manfred Reichert. 2020. Robotic Process Automation–A Systematic Literature Review and Assessment Framework. *arXiv preprint arXiv:2012.11951* (2020).

[62] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[63] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268.

[64] Dell Zhang, Alexander Kuhnle, Julian Richardson, and Murat Sensoy. 2020. Process Discovery for Structured Program Synthesis. *arXiv preprint arXiv:2008.05804* (2020).

[65] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.