Improving Crowd-Supported GUI Testing with Structural Guidance

Yan Chen, Maulishree Pandey, Jean Y. Song, Walter S. Lasecki, Steve Oney
University of Michigan
Ann Arbor, MI, USA
{yanchenm, maupande, jyskwon, wlasecki, soney}@umich.edu

ABSTRACT

Crowd testing is an emerging practice in Graphical User Interface (GUI) testing, where developers recruit a large number of crowd testers to test GUI features. It is often easier and faster than a dedicated quality assurance team, and its output is more realistic than that of automated testing. However, crowds of testers working in parallel tend to focus on a small set of commonly used User Interface (UI) navigation paths, which can lead to low test coverage and redundant effort. In this paper, we introduce two techniques to increase crowd testers' coverage: *interactive event-flow graphs* and *GUI-level guidance*. The interactive event-flow graphs track and aggregate every tester's interactions into a single directed graph that visualizes the cases that have already been explored. Crowd testers can interact with the graphs to find new navigation paths and increase the coverage of the created tests. We also use the graphs to augment the GUI (GUI-level guidance) to help testers avoid only exploring common paths. Our evaluation with 30 crowd testers on 11 different test pages shows that the techniques can help testers avoid redundant effort while also increasing untrained testers' coverage by 55%. These techniques can help us develop more robust software that works in more mission-critical settings, not only by performing more thorough testing with the same effort that has been put in before, but also by integrating these techniques into different parts of the development pipeline to make more reliable software in the early development stage.

Author Keywords

GUI testing; Software testing; Crowdsourcing

CCS Concepts

•**Human-centered computing** → **Human computer interaction (HCI); Interactive systems and tools;** User studies;

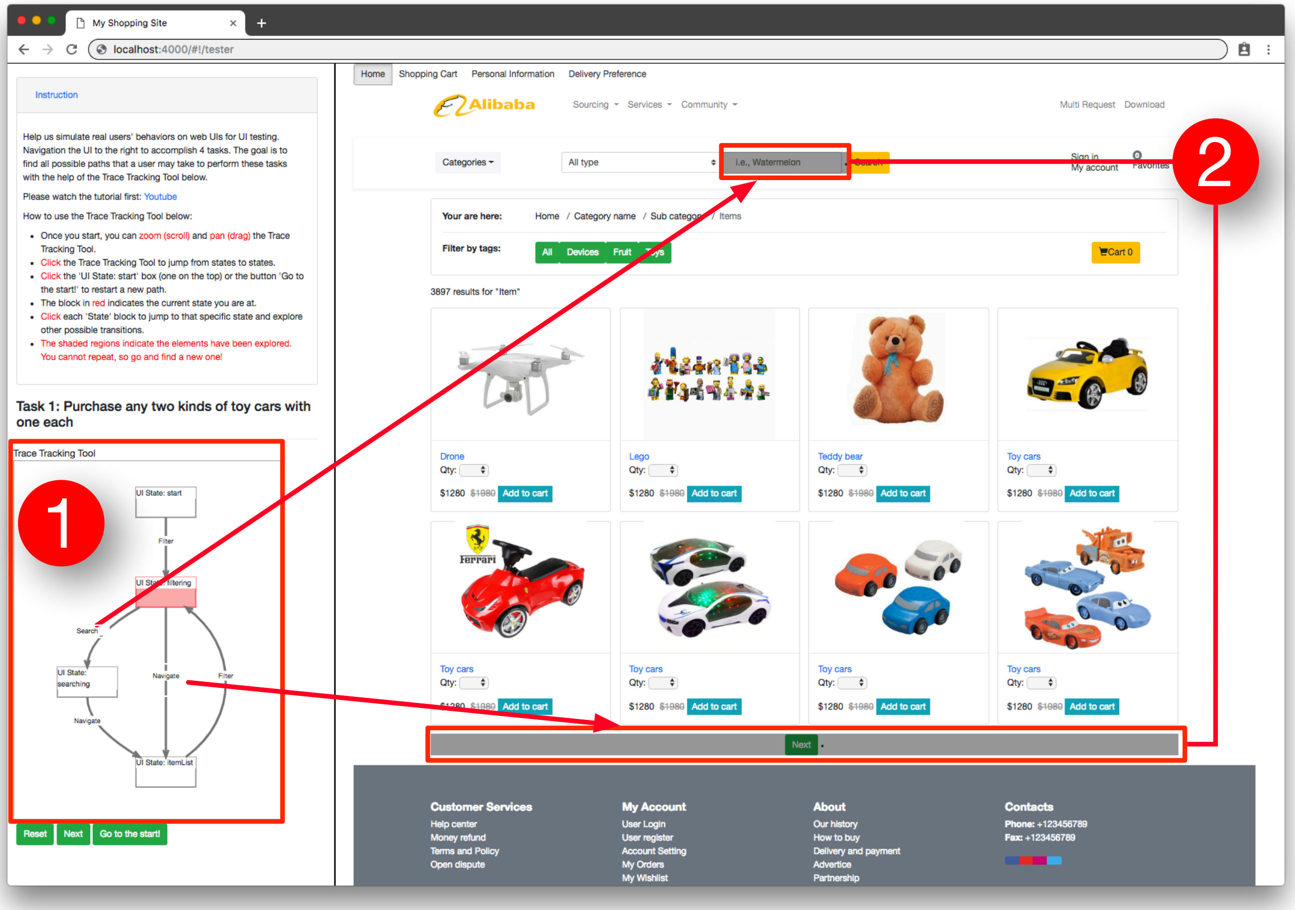
CHI '20, April 25–30, 2020, Honolulu, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376835

INTRODUCTION

Software testing is an important, yet often overlooked, part of the software development lifecycle. In the case of GUI development, testing helps developers find functional and usability defects in a system's front end. This testing requires test cases that consist of a sequence of input events (e.g., writing in the input field and then clicking a button), which we define as *navigation paths*, and the resulting output (e.g., a modal window pops up) [4, 54], which we define as *GUI state*. Prior work has shown that GUI testing can be effective in finding both front-end and back-end defects because they reflect usage scenarios and often execute back-end code [8, 43]. However, due to the multitude of possible user event sequences, it can be challenging to design a comprehensive set of tests even for simple user scenarios (e.g., purchasing an item on an e-commerce site).

Traditionally, software testing was conducted by dedicated quality assurance (QA) teams with formally trained testers. Although these QA teams are reliable, the high cost and delayed responses made them hard to scale and non-flexible for rapid update needs for the software industry today. Automated testing could be one solution, but the inability to create realistic user behavior test cases makes them hard to rely on given the variations in software products *Crowd testing* is an emerging practice that enables testing with more flexibility and scalability than QA teams [15, 27, 48, 49, 50]. It involves recruiting crowd workers (either untrained or trained) from platforms like Mechanical Turk [2] or uTest [3] to perform GUI tests. However, crowd testing often results in a high degree of test case duplication [49], because crowd workers tend to navigate the same common paths while working in parallel. Prior work focused on analyzing workers' responses to identify and remove duplicates [49], rather than preventing the issue. This duplication of test cases can lead to lower test coverage, making the testing process less effective or more costly.

To address this duplication problem, our insight is to augment GUI testing with visual cues that guide testers' attention to unexplored navigation paths. Specifically, we propose *interactive event-flow graphs* and *GUI-level guidance* (Fig. 1), to make crowd testing more effective. These techniques give testers a human-readable navigation path history graph that is situated on testers' current GUI state. This draws on information foraging theory [29, 40, 52], which suggests that providing



Help us simulate real users' behaviors on web UIs for UI testing. Navigation the UI to the right to accomplish 4 tasks. The goal is to find all possible paths that a user may take to perform these tasks with the help of the Trace Tracking Tool below.

Please watch the tutorial first: Youtube

How to use the Trace Tracking Tool below:

- Once you start, you can zoom (scroll) and pan (drag) the Trace Tracking Tool.
- Click the Trace Tracking Tool to jump from states to states.
- Click the 'UI State: start' box (one on the top) or the button 'Go to the start!' to restart a new path.
- The block in red indicates the current state you are at.
- Click each 'State' block to jump to that specific state and explore other possible transitions.
- The shaded regions indicate the elements have been explored. You cannot repeat, so go and find a new one!

Task 1: Purchase any two kinds of toy cars with one each

Figure 1. The interactive event-flow graph (1) shows testers' traces in real time. Testers can go to any previously explored states by clicking the state node. The graph also derives the GUI-level guidance (2) that adds a non-clickable CSS overlay on the previously explored DOM elements to prevent duplicate activity. Currently, this GUI is at the "filtering" event, indicated by the event node with red color in the graph. The red arrows show the overlays are consistent with the outgoing event-flows of the "filtering" event.

effective visual cues can lower users' effort in finding the desired information. This tight coupling of testers' navigation path history (both their own and others') to UI elements, in combination with human-readable interactive graphs, is central to our approach to reducing the redundancy of test cases and encouraging new path discovery.

To validate their effectiveness in guiding testers, we instrumented 11 realistic GUIs with the interactive event-flow graphs and GUI-level guidance, and conducted a between-subject experiment involving 30 participants with different levels of expertise. We measured their performance using GUI test coverage metrics [33]. The 330 test scripts that were generated suggest that testers, regardless of their prior expertise, can use our techniques to significantly improve their performance in event-interaction coverage by avoiding duplicates. Our techniques shaped testers' working strategy such that they would not waste their effort on repeated work, but concentrate on creating new test cases by making use of their prior experience and seek new ways to "break the application."

We make the following contributions in this paper:

- a new approach instantiated by two techniques, GUI-level guidance and interactive event-flow graphs, which visually guide workers using GUI underlying structures (e.g., DOM tree) toward more effectively testing GUIs; and
- experimental results that show these techniques can help testers find more event-transitions and avoid duplication.

BACKGROUND & RELATED WORK

GUI testing has become an important step in the software development lifecycle because GUIs are the primary UI in the vast majority of today's commodity software [32, 39]. However, creating GUI tests to cover the large number of possible user event sequences is a significant challenge.

Automated Testing

Manual testing can be labor-intensive and expensive. For instance, it is hard to expect developers to perform an in-depth GUI test on every commit. Some companies employ a dedicated tester team per product; however, it is hard to quickly scale the number of testers up or down in response to changes in demand (e.g., to continuously test a new experimental branch of the product). To address the challenge of covering a large state space, prior work has developed automated testing techniques to generate [7, 39, 47] and execute test cases [14, 55] at scale. Despite these techniques, empirical research [18, 41] has shown that companies still rely on manual testing because test execution is not a simple mechanical task but a creative and experience-based process. A survey of software developers showed that 94% of the 115 respondents agreed that manual testing will never be replaced by automated testing [41]. However, these techniques require tedious configuration, created test cases can easily break due to even minor changes in the GUI [15], and generated event sequences are often not representative of user event sequences in the real world, resulting in low coverage [28].

Crowdsourcing GUI Testing

Crowd testing is an emerging trend in GUI testing [15, 27, 48, 49, 50], where GUI developers recruit testers from platforms like Mechanical Turk [2], Baidu Crowd Test [6], or uTest [3]. Other prior work on crowd testing has found benefits (e.g., low cost and ease of tester recruiting), but it has some drawbacks as well. Most notably, because most crowd testing tools do not share awareness of explored paths among testers, crowd testing can produce many duplicates, leading to wasted effort for both developers and testers [49, 53].

There are two primary categories of GUI testing: functional testing and usability testing. Functional testing helps ensure the GUI works according to specification, regardless of how usable that specification is. Usability testing helps ensure users are able to use the GUI effectively. Our techniques are designed for functional testing, and thus we aim to create traces that can put the GUI into as many states as possible in order to find functionality bugs. ZIPT [13] has explored ways to improve crowdsourced usability testing by collecting, aggregating, and visualizing users' interaction paths with mobile applications. Thus, unlike our techniques, ZIPT does not prevent users from creating duplicate test cases because understanding which interaction paths are the most common is helpful for assessing a GUI's usability.

Prior work on collaborative crowdsourcing has introduced techniques that make crowd workers aware of prior responses to generate more diverse answers [9]. Legion [23] automatically proposes a previously used label for actions in videos to prevent crowd workers from always generating new ones for each occurrence, which makes the labels highly consistent. In the context of GUI testing, a common method is to remove duplicates from the list of test cases by a post-hoc result analysis [49]. Other researchers have proposed incentive-based approaches that reward testers who discover previously unseen cases [5]. While these approaches help reduce duplicate tests, the suboptimal efficacy of crowd testers remains because many of them will produce test cases that are later removed as duplicates. Our proposed techniques for improving testers' efficacy is inspired by the ExtraSensory Perception (ESP) game [46]. Similar to the game's "taboo" mechanic, our techniques indicate which actions have already been explored. However, instead of asking participants to guess existing answers, we ask them to find cases that are different from the existing ones.

Inferring Task Models from Interaction Traces

Prior work has used crowd testing to generate task models, which are then fed into automated test generators. For example, SwiftHand [10] learns models of Android applications and uses them to find unexplored states. MonkeyLab [25] models user event interaction sequences on Android applications to generate new test cases. POLARIZ [28] simulates user interaction patterns learned from users' behavior on Android apps, and then it applies this simulation to different applications. Rico [12] proposed a hybrid approach that records crowd workers' app traces first and then continues the exploration programmatically, reaching a wider state space in an app. These approaches combine the advantages of humans and machines, making test cases realistic and testing tasks scalable. However, the issue of test duplication remains.

Techniques for inferring interaction models from prior usage data have also been proposed [8, 16, 36]. Brooks and Memon ([8]) inferred a probabilistic model of user behavior; Ermuth and Pradel ([16]) inferred a deterministic model; and Fard et al. ([36]) inferred a model per task. All of this prior work has inferred task models from crowd workers' and users' natural interaction traces. By contrast, our goal is to actively guide crowd workers away from common interaction traces to find paths that are less common.

Improving GUI Tester Efficacy

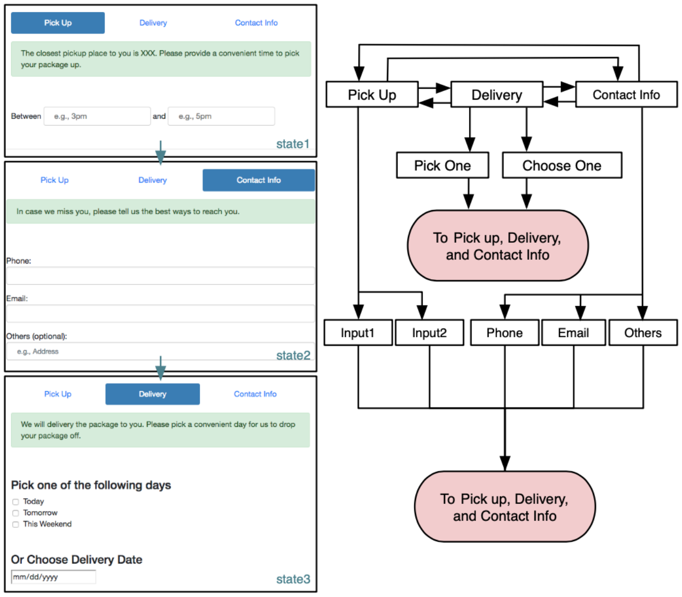
Micallef et al. [35] showed that they can improve the performance of untrained GUI testers by giving them a summary of common testing strategies derived from best practices [51]. Instead of providing testers with general testing tips, our interactive event-flow graphs and GUI-level guidance techniques give testers *in situ* guidance to lower the cognitive effort for their decision-making. Other work, like MOOSE [11] and COCOON [53], studied improving testers' efficacy by optimizing the tester hiring process. Such techniques could be used in combination with those proposed in this paper, but improving the hiring process is outside of the scope of this paper.

DESIGN GOALS AND RATIONALE

Designing a UI that facilitates effective GUI testing is challenging because testers could take many navigation paths to complete a task (e.g., there are many possible event sequences that one could take to purchase an item on an e-commerce site) and some of the paths might have overlapping sub-paths. The convoluted navigation paths can make it difficult for testers to remember where they already navigated and where else they could navigate to increase the testing coverage.

Reducing Overlapping Navigation Paths

To reduce overlapping and redundant sub-paths in testing, we implemented GUI-level guidance that presents previous workers' navigation paths to new workers by augmenting the UIs with non-clickable CSS overlays (Fig. 1 ②). We designed this GUI-level guidance by conducting a series of small studies, comparing two different approaches of presenting GUI navigation paths: (1) UI overlays that block out regions in the UI


The closest pickup place to you is XXX. Please provide a convenient time to pick your package up.

Between e.g., 3pm and e.g., 5pm

In case we miss you, please tell us the best ways to reach you.

Phone:

Email:

Others (optional): e.g., Address

We will delivery the package to you. Please pick a convenient day for us to drop your package off.

Pick one of the following days

□ Today
□ Tomorrow
□ This Weekend

Or Choose Delivery Date

mm/dd/yyyy

Figure 2. An Event-Flow Graph (EFG) for the "delivery preference" page of an e-commerce website. The left column shows three different states of the GUI, differing based on which tab on the top of the page is clicked. The right column shows the event-flow graph of the GUI where the top three nodes of the graph each represent different tabs on the left column. With the event-flow graph, testers can easily navigate and discover feasible paths, eventually increasing the testing coverage.

that have previously been explored, and (2) textual logs that show past user events. We chose to compare these presentations because prior work showed that presenting previous people's responses can effectively improve others' task performance [19, 29, 46]. However, unlike their approaches, we focused on guiding testers to avoid explored GUI regions, thus including information about how *often* users use particular UI features (such as a heatmap) was unnecessary and potentially misleading.

We compared two presentations with a baseline, which was not presenting any guidance to the workers. We measured the path duplication rate after testers used the two presentations and found that testers could successfully avoid repeating previously explored navigation paths with the UI overlay guidance, but they could not in either the baseline or the textual log condition. Participants reported that there were mainly two advantages of having the overlays on the UI. First, overlays required less context switch to look at which paths were already explored. Second, overlays required less navigation effort when making the decisions on which path to explore next. Therefore, we decided to use the non-clickable CSS overlays as our GUI-level guidance.

Increasing Test Coverage

To encourage testers to efficiently increase the test coverage, the UI should enable them to easily navigate among previously explored events to find a broader range of test cases. Prior work has suggested two models to represent previously explored interactions: Finite State Machines (FSMs) [34, 47], and Event-Flow Graphs (EFGs) [39]. However, a study [47] suggested that these approaches can be overwhelming for testers to evaluate because the number of possible permutations of low-level events and targets are too large to test, especially when the context of the path is missing. So developers typically rely on manually crafting a small number of event sequences, which is not scalable.

Inspired by prior work that developed an abstract GUI model [31], we design an interactive, abstract EFG as part of our guidance techniques, in which testers can easily understand and navigate the graphs by a simple click interaction on a node (Fig. 1 1). With our interactive event-flow graphs, clicking an event node lets the GUI return to the event-associated state. For example, clicking the node of "filtering" event in Fig. 1 2 will set the GUI to the results page of the filter button being clicked. The active (testers' current) event of the EFG also updates to the "filtering" event. One can use many kinds of techniques to decide which filter buttons should be clicked, such as applying the values from the last event that occurred on the GUIs. Because we focus on increasing testers' event-flow coverage, we used a set of predefined values for each state.

APPROACH AND IMPLEMENTATION

In this section, we introduce the implementation of the two novel crowdsourcing techniques for efficient GUI testing: GUI-level guidance and interactive event-flow graphs.

The GUI-level guidance

To help avoid duplicate test cases, we propose using GUI-level guidance that displays information about existing test cases by augmenting the GUI (Fig. 1 2). We implemented this guidance by adding a gray CSS overlay on top of the explored elements. The overlay can prevent testers from interacting with elements that lead to previously explored interaction paths, encouraging them to find other widgets to explore, and helps generate more effective outputs.

The interactive event-flow graphs

To visualize the explored interaction paths, we built a human-readable Event-Flow Graph (EFG) to represent the event-flow of the Application Under Test (AUT) by incorporating previous testers' traces. Figure 2 shows an example of an EFG for a UI for specifying a user's delivery preferences on a representative e-commerce website. At the top are three nodes (or events), Pick Up, Delivery, and Contact Info, which represent the delivery option tabs. They are clickable navigation "buttons" that are available when the delivery preference page is first invoked. The edges represent the event flows (or event-interactions) from node to node. To measure the effectiveness of GUI testing using EFGs, prior work developed coverage criteria that calculate the number of event-interactions within all the generated event sequences [33]. The number of event-interactions for a single event, such as "Pick Up" in Fig. 2, is calculated by counting all the outgoing event-flows (arrows) of this node (i.e., "Pick Up" → "Between `Input1`," "Pick Up" → "Between `Input2`," "Pick Up" → "Delivery," and "Pick Up" → "Contact Info"). We use the same criteria to measure the effectiveness of our proposed techniques.

As discussed in the related work section, the number of possible event sequences for a GUI can be enormous, making the corresponding EFG difficult for testers to understand and interact with. To address this problem, our techniques allow end-user developers to abstract the meta-level events (e.g., click the "Today" checkbox) to a user-intent-level (e.g., pick a delivery day). We did this by instrumenting the parent nodes in the DOM tree instead of an individual leaf node (i.e., nodes without any children). This provides crowd testers an easy way to read navigation history in the EFG.

To track testers' traces, we implemented a tracker on the client side. We did this by creating an empty EFG object, developed based on the Dagre libraries [1], so that events and event-flows can be added to it in event handlers JavaScript function on the client side (Fig 1. ②). To tailor the user event displayed on each node to a human-readable level, we presented the corresponding user intents. These user intents come from the unique attribute values of the parent nodes that end-user developers are assigned to, so that when testers are interacting with their children nodes they are automatically be triggered.

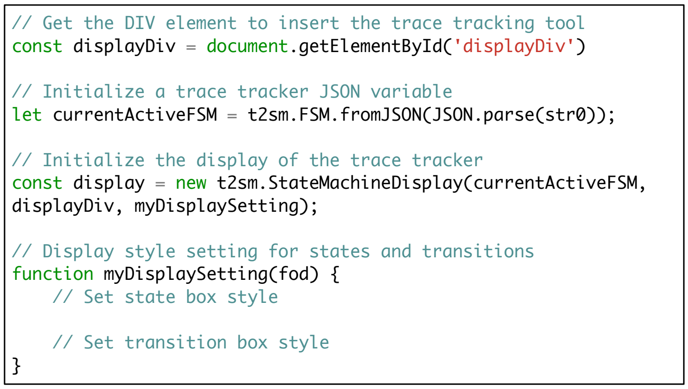
The event name displayed on each node could come from any attributes of the corresponding parent node (e.g., id='delivery_day_checkboxes'). To decide whether to add a new event-flow to the EFG or to fire an existing one, the tracker compares the incoming event to all the existing event-flows of the current active event and makes the decisions. Because the interactive event-flow graphs run in real time while testers perform their tasks, we used a computationally inexpensive method. Our evaluation shows that this approach is effective. Note that exploring advanced trace tracking techniques (e.g., quadtree decomposition [42] or element association analysis [12]) is beyond the scope of this work on studying effective tester guidance.

Implementation

Our technique is a JavaScript library that was developed based on the Dagre libraries. To instrument a standard web application, one can create a tracker instance (Fig. 3, variable name `currentActiveFSM`), and listen to events to the desired parent nodes by adding unique ID values to them. The output is an array of JSON objects, which we saved using Firebase.

EVALUATION OF GUIDANCE TECHNIQUES

To evaluate our crowd-powered GUI testing guidance techniques (GUI-level guidance and interactive event-flow graphs), we conducted an experiment in which crowd testers were given 11 GUI testing tasks and were asked to perform them on three web GUIs that were instrumented with both techniques. To make the EFG human-readable, we instrumented their DOM trees so that each event in the EFG denoted one type of user event (e.g., filtering) that was associated with a group of DOM elements (e.g., all "filter" buttons), and each transition denoted the immediate transition action from one event to another. Although this design is different from the standard approach, we hypothesized that our approach can effectively help testers easily navigate through previous traces by providing a readable and scalable EFG. We tested this both individually and collectively (i.e., building on top of an existing EFG generated by others) performing the tasks on a web GUI prototype. We refer the integration of these two techniques as the "guidance" throughout this section. In this section, we first talk about our study setting. Then, we discuss our study results and analysis.



```
// Get the DIV element to insert the trace tracking tool
const displayDiv = document.getElementById('displayDiv')

// Initialize a trace tracker JSON variable
let currentActiveFSM = t2sm.FSM.fromJSON(JSON.parse(str0));

// Initialize the display of the trace tracker
const display = new t2sm.StateMachineDisplay(currentActiveFSM,
displayDiv, myDisplaySetting);

// Display style setting for states and transitions
function myDisplaySetting(fod) {
    // Set state box style

    // Set transition box style
}
```

Figure 3. Example of Javascript code that creates a tracker instance to instrument a GUI.

GUI Testing Tasks

We wanted to ensure our study's GUIs and tasks were realistic. We also wanted to test the type of websites that are commonly used, so we chose three common categories: travel agent, blog, and e-commerce (see Fig. 4 for screenshots of all the GUIs). Because our techniques require instrumenting the GUI code, we also needed to have access and permission to modify the source code for the GUIs we used. We synthesized the necessary data (e.g., product information) to make it more realistic (further validation is in the discussion).

To get realistic tasks for testing, we recruited an independent professional tester with five years of experience of web application quality assurance from Upwork. We gave the tester the three aforementioned types of websites and asked them to design and make GUI testing tasks. We eliminated tasks that required checking the content or sanity checks (e.g., "*Review the content of the article XXX*") because they often require domain knowledge that crowd testers might not have. Additionally, because our techniques focus on event sequences, we excluded the tasks regarding cross-device compatibility (e.g., "*Check cross-browser compatibility*"). We ended up with the following 11 tasks, which we used for the final evaluation:

- **Travel Agent**
  - Task 1: Find an Asian restaurant and reserve the place
  - Task 2: Find an Indian buffet that allows dogs and accepts Visa
  - Task 3: Write, edit, and rate a review
- **Blog**
  - Task 4: Find an article about culture and bookmark it
  - Task 5: Read an article and bookmark it
  - Task 6: Write/edit/bookmark an article
  - Task 7: Discuss the article with its author
- **E-commerce**

[1] https://github.com/dagrejs

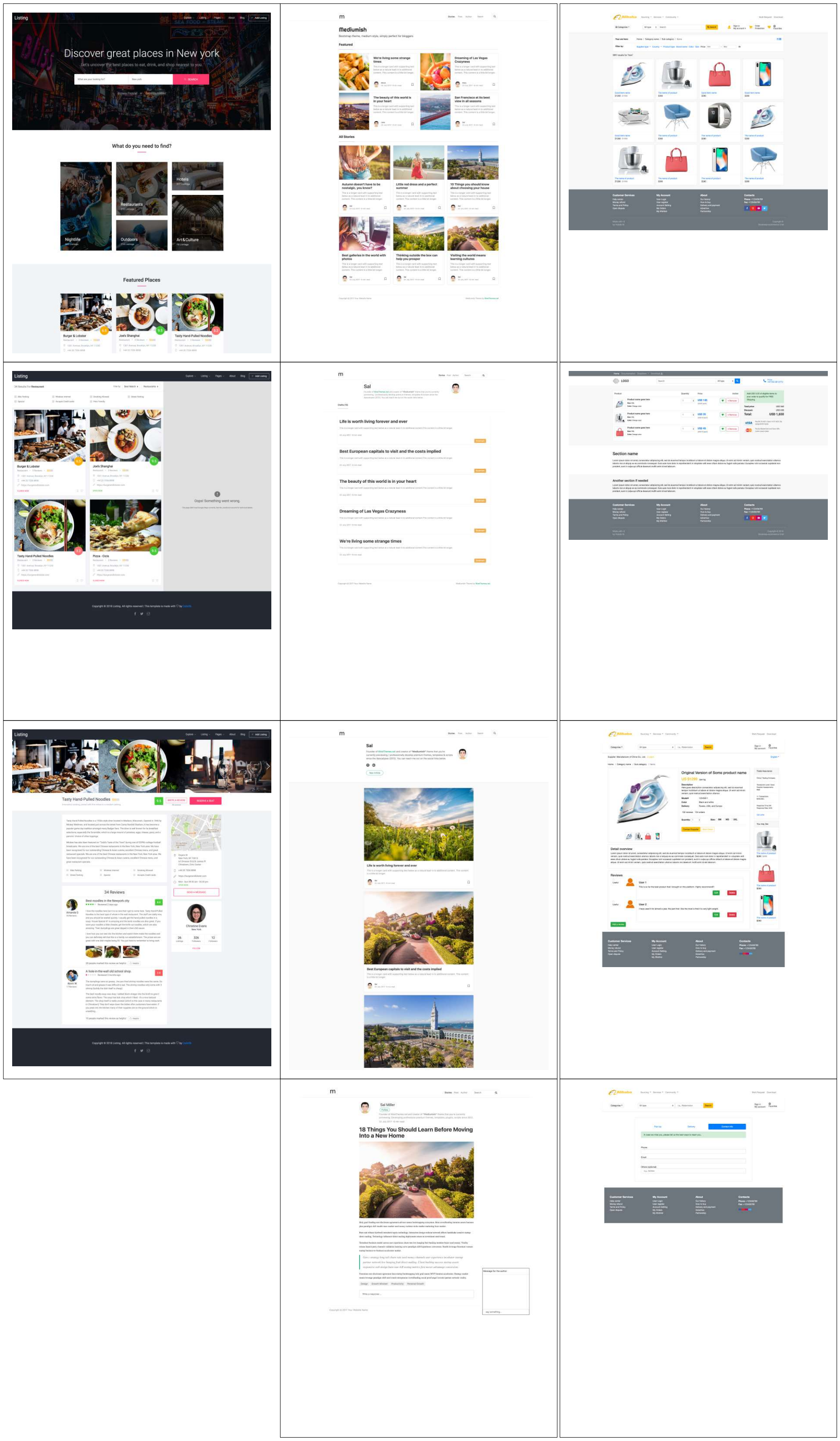

Listing
Discover great places in New york
What do you need to find?
Hotels
Restaurants
Nightlife
Outdoors
Art & Culture
Featured Places
Mediumish
Featured
All Stories
Sal
Life is worth living forever and ever
Best European capitals to visit and the costs implied
The beauty of this world is in your heart
Dreaming of Las Vegas Crazyness
We're living some strange times
Section name
Tasty Hand-Pulled Noodles
34 Reviews
Original Version of Some product name
18 Things You Should Learn Before Moving Into a New Home

Figure 4. Screenshots from our three GUI prototypes. From left to right columns: travel agent, blog, e-commerce

- Task 8: Find a toy and add it to the shopping cart
- Task 9: Change my shopping list and make the total price lower than $5 USD
- Task 10: Add/edit/rate a review
- Task 11: Verify all the delivery methods

Although prior work has used artificial defects in their system under testing [35], all of the 11 GUIs we used were bug-free to avoid potential biases. This also allowed us to evaluate the effectiveness of the tool rather than the testers' expertise.

Participants

We recruited 30 participants: 18 untrained testers from MTurk and 12 trained testers from Upwork. The 18 MTurk participants were crowd testers who had a minimum of 90% acceptance rate and finished at least 500 Human Intelligence Tasks (HITs). The 12 Upwork participants all had at least one year of experience in manual GUI testing. MTurk participants were compensated at $8.00 per hour, and Upwork participants were compensated at $18.00 per hour. At the beginning of each session, participants were asked to watch a short tutorial video and familiarize themselves with the application. We also conducted a follow-up survey among the trained testers regarding their experiment experience.

Experimental Design

Our study had five conditions, each with six testers. The 11 tasks described in the previous section were used in all conditions. Our conditions permute combinations of *untrained* (U) or *trained* (T) testers, and *guidance* (G) or the *baseline* (B):

- C_{UB} (untrained baseline): untrained testers / no guidance,
- C_{TB} (trained baseline): trained testers / no guidance,
- C_{UG} (untrained with guidance): untrained testers / guidance,
- C_{UG+} (untrained collaborative with guidance): untrained testers collaborate with each other using guidance,
- C_{TG} (trained with guidance): trained testers / guidance.

For C_{UB} and C_{TB}, we gave participants the task description, the study goal (i.e., find all possible traces to accomplish the tasks), and the three testing GUIs. At any point, they could go back to the initial event to restart their navigation (Fig. 1.1 "Go to the start!" green button) or move on if they thought they had found all the traces. The same information and setup was provided in C_{UG}, C_{UG+}, and C_{TG}, but these groups had the guidance enabled. To evaluate the guidance's effectiveness in supporting collaboration, testers in C_{UG+} were given one of the C_{UG} testers' EFGs, which could have been already fully covered, and they were instructed to build on top of it to accomplish the same navigation tasks. All the EFGs generated in C_{UG} were paired with a tester in C_{UG+}. In total, the study yielded 330 (6 workers per condition × 11 tasks × 5 conditions) data points (sets of GUI-level activities for a task).

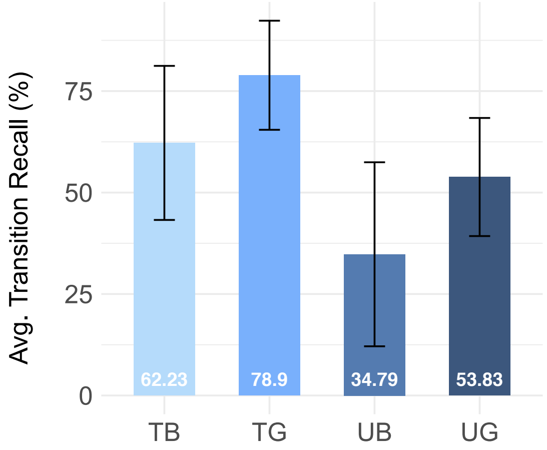


Figure 5. Average transition coverage for four single worker conditions based on these combinations: trained baseline (TB), trained with guidance (TG), untrained baseline (UB), untrained with guidance (UG). A higher transition coverage is better.

Coverage Metrics

GUI testing requires its own set of metrics to evaluate the effectiveness of a test. This is because GUIs are *event-driven*—their behavior is defined by how they react to user and system events. These event callbacks often reference and modify a shared state. As a result many bugs occur when callbacks make invalid assumptions about the application state, often because callbacks were executed in an order the developer did not anticipate [37]. Traditional code coverage metrics, which measure *how much* of the code was executed, have been found to be poor metrics for evaluating the effectiveness GUI tests [30, 33]. Code coverage measures *whether* a given piece of code was executed whereas GUI tests should focus on *how many feasible states* a piece of code was executed in. In other words, we consider *state* coverage to be more important than *code* coverage.

It is infeasible to determine the precise state coverage of GUI tests because most realistic GUIs have too many possible states. Instead, prior work has proposed metrics to approximate state coverage. One such metric is event-interaction coverage, which examines how many permutations of input events have been tested [33]. We adopted this metric to evaluate the tests' effectiveness by measuring their coverage relative to the "ground truth," the EFG that our researchers manually crafted.

Performance Metrics

For each task, we measured the event-interaction coverage, repetition, and transition discovery time by the following means:

- Event-interaction coverage: (number of discovered event-interactions) / (number of all possible event-interactions),
- Repetition: number of fired event-interactions (or number of discovered unique event-interactions), and

- Transition discovery time: (average time spent per task) / (number of discovered unique event-interactions).

These measurements indicate the overall effectiveness of the guidance given the set of tasks on the testing GUI. Because the name of an EFG node comes from its corresponding element ID, we crafted the element ID in the GUI DOM tree to make nodes easy to understand for the testers in C_{UG} and $C_{\text{UG+}}$. Although the names of the EFG nodes are non-trivial to derive in real-world sites (e.g., dynamic name convention), we believe that using widget icons/images to indicate the transition actions would also be effective for presenting traces.

Results

In this section, we discuss the guidance's transition coverage (percentage), transition discovery efficiency (time), and repetition (occurrence) across different conditions. To measure statistical significance, we ran a pairwise two-tailed t-test and Welch's t-test (unequal variances).

The guidance eliminated test case duplication

Without the guidance, we found that untrained testers repeated 43.94% (standard deviation $\sigma = 24.63\%$) of their own transitions, and trained testers repeated 45.75% ($\sigma = 11.22\%$) of theirs. We also found that pair collaboration for both untrained and trained testers generated duplicate transitions with 43.94% ($\sigma = 24.63\%$) and 45.75% ($\sigma = 11.22\%$) occurrence rates on average. In contrast, none of the testers using the guidance (C_{UG}, $C_{\text{UG+}}$, C_{TG}) generated duplicate transitions, indicating that the guidance can robustly prevent testers from interacting with previously explored elements. By further analyzing testers' GUI element-level interactions, we found that trained testers (C_{TB}) often spent their effort on testing widgets within the same abstract state. For instance, when testing Task 6, P3 from C_{TB} generated more than five traces that were combinations of elements in the "Editing Article" state and the "Bookmarking Article" state. The only difference is the selected article, which can be automatically chosen once the event sequence is determined. This high duplication rate has been reported by prior work [49].

The guidance improves trained & untrained testers' coverage

Figure 5 shows the overall performance across different conditions. We found that untrained testers using the guidance (C_{UG}) covered significantly more transitions than those without (C_{UB}), resulting in coverage of 34.79% (C_{UB}) and 53.83% (C_{UG}) ($p < 10e^{-7}$), respectively. Furthermore, trained testers using the guidance (C_{TG}) also covered significantly more transitions than those without (C_{TB}), resulting in coverage of 62.23% (C_{TB}) and 78.90% (C_{TG}) ($p < 10e^{-7}$), respectively. These coverage improvements indicate that the guidance is effective in guiding testers, regardless of their expertise, to discover more transitions than they could without the guidance.

Trained testers without guidance (C_{TB}) still outperform untrained testers with guidance ($C_{\text{UG+}}$)

Comparing the average transition coverage of untrained testers with guidance (C_{UG}) to that of trained testers with guidance (C_{TG}), the results showed statistical evidence that seasoned testers can outperform untrained testers (53.83%, 78.90%, $p < .005$). It also makes sense, because a tool cannot suddenly help untrained testers perform as well as trained ones.

	Average time(s)	Time(s) per transition
C_{UB}	393.07 (163.90)	32.54 (18.86)
C_{UG}	439.37 (469.37)	19.71 (20.40)
$C_{\text{UG+}}$	301.05 (189.96)	9.50 (5.71)
C_{TB}	525.70 (286.60)	20.06 (8.92)
C_{TG}	430.71 (402.86)	11.62 (8.63)

Table 1. The average time participants spent per condition, and the time it took them to discover a new transition.

The guidance improves trained testers' discovery speed

We computed the average task completion time and the average time it took a participant to discover a new transition for all conditions (Table 1). Our results indicate that the guidance has no statistical impact on untrained testers regarding both time metrics ($p > .058$, $p > 0.93$, respectively). Similarly, we did not find a statistical difference between the time trained testers spent when using the integration and without using it to complete the task ($p > 0.42$). However, the time that trained testers took to discover a new transition is shorter when they used the guidance ($p < 0.0068$), indicating that the guidance makes the trained testers' performance more efficient. We suspect that this time reduction was only apparent with trained testers because the EFG matched their mental model of test creation and could serve as a memory aid. We believe that the untrained testers' benefits were less pronounced because they were less familiar with how to strategically use EFGs.

The guidance helps testers collaborate

We simulated pair collaboration in C_{UB} by calculating the union sets of transitions generated by all pairs of testers ($P(2, S_{C_{\text{UB}}})$). Compared to $C_{\text{UG+}}$, we found that untrained testers using guidance ($C_{\text{UG+}}$) could collaborate and improve the coverage significantly (Welch's t-test, $p < .0001$), with average transition coverages being 36.16% and 71.39%. This indicates that the guidance could support pair collaboration to improve transition coverage. Additionally, it shows that untrained testers did not explore many of the new events without guidance.

DISCUSSION

In summary, we found that the combination of the GUI-level guidance and interactive event-flow graphs can effectively guide both untrained and trained testers to significantly increase their event-interaction coverage. Consistent with Information Foraging Theory [40], this finding suggests that providing visual navigation cues could help guide people's attention and thus improve information access.

Testers' experience with our techniques

Overall, testers found the guidance "quite helpful to find paths and avoid duplicate actions" (P1) and "user friendly" (P4), and felt that "all scheme [was] forming just right on your eyes" (P5). One participant said "[the guidance] helped me to save my time and explore the new links without clicking the already

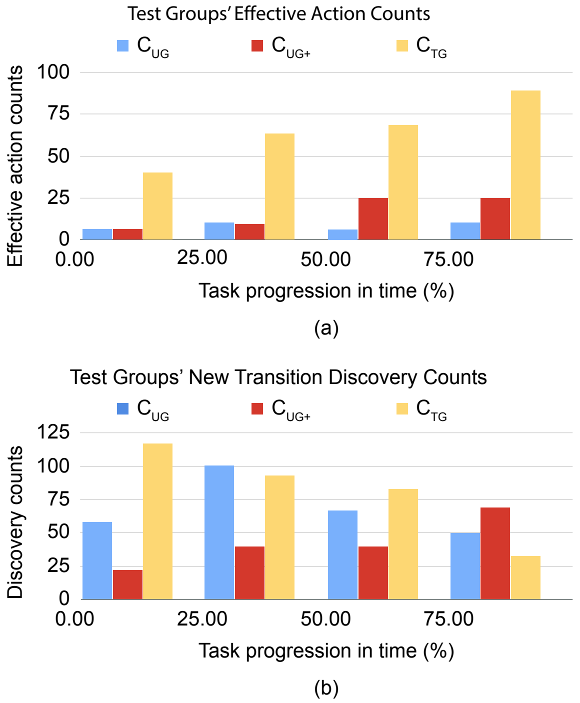


Figure 6. Testers' interaction pattern with respect to task progression in time. Chart (a) shows trained and untrained testers' guidance graph click counts. As time progressed, testers interacted with the guidance graph more because it was harder to find a new transition discovery. Chart (b) shows the number of new transition discovery counts, which decrease as time progresses, implying the discovery becomes more challenging (thus more clicking happening in (a) as time progresses).

explored link" (P1). Testers also suggested a few ideas for improving the guidance. P11 said that it would be nice to enable a transition- or node-removal function, allowing testers to better focus on expanding major paths. This makes sense, given that the task scopes can be large enough to include a considerable number of paths. In this case, presenting all explored paths could complicate the interactive event-flow graph, making it less useful in terms of finding new transitions. Another tester suggested having a trace log to "record all user doings in log format" (P6). Some other testers did not immediately realize that they could jump to previously discovered events by clicking the nodes; thus, they wasted some time before understanding this function.

The study materials were considered realistic

In our post hoc survey, all 12 trained testers thought the 11 tasks were "reasonable" (P1,2,3,5,11), "realistic" (P4,8), "good example to analyze module relationships" (P6), and "covered the basic of all the websites" (P7). Also, 10 of them thought that the GUIs were "a normal website" (P1), "not different from other testing jobs" (P6), and "realistic" (P2,4,5,8,9,12). Two testers thought the websites were not realistic because "some functions are not working properly" (P10), and some have "bugs" (P11). But, P10 then pointed out that "it is okay if you just need to check the path."

The usage of the guidance increased throughout the task

To further evaluate how much the guidance helped testers throughout the study, we measured the usage of the interactive event-flow graphs using *effective action*, a user action where its previous action is a click on the graph and the current action is a new GUI state. We chose to measure effective action because we believe that the clickable feature in an EFG could potentially save testers' effort of navigating back and forth using the testing GUI by directly jumping to any discovered states. We calculated the average number of effective actions per worker for the three groups that used the guidance: C_{UG} ($\sigma = 3.72$), C_{UG+} ($\sigma = 5.50$), and C_{TG} ($\sigma = 14.56$). Furthermore, we projected these numbers to the normalized task time, shown in Fig. 6(a), and revealed that, for all the conditions, the average number of usage had been increasing as testers performed the tasks. Meanwhile, the number of new discovered transitions had been decreasing as time progressed, as shown in Fig. 6(b). One tester described their strategy as "make [the] basic traces for a common user, then make the possible combinations." This indicates that guidance aids testers in finding less common transitions, which helps explain why testers in these groups had higher coverage.

'The guidance shaped my testing strategy.'

When unpacking individual testers' traces, we found that the interactive event-flow graphs shaped testers navigation patterns. Using Fig. 2 as an example, we found that without the guidance, testers had more single-thread navigation traces, such as "Pick Up" → "Input1" → "To Pick Up, Delivery and Contact Info," whereas with the guidance testers explored multiple sub-paths to maximize the exhaustion, such as "Pick Up" → "Input1" → Go back to "Pick Up" using the graph → "Input2." Testers were also able to do this without the interactive event-flow graphs, but the cases were less frequent.

In our follow-up interview, we also found that all six trained testers felt the guidance changed their testing strategies from their prior approach and helped them better plan their moves at given states. The six trained testers in the C_{TG} condition used the guidance as their extended memory to "avoid the same trace" (P5), and "save my time and explore the new links" (P1). This implies that guidance could aid individual testers' memory for personal information management. Prior work has found that untrained testers often conduct GUI testing without strategies [35]. We imagine that in the future, we can present these strategies to guide untrained testers to increase their performance in coverage and scale the GUI testing process.

Scalability

Both techniques (interactive event-flow graphs and GUI-level guidance) are computationally inexpensive and could thus easily be scaled to real-world applications. Thus, the primary type of scalability that we consider is the ability to scale to complex EFGs. Because a graph is tied to a sequence of user intents for completing a task, theoretically tasks that yield dynamic user intents and require more steps to complete would result in more visually complex graphs, which can be difficult for testers to make use of [44]. However, prior work has empirically shown that only 14 unique interaction patterns were needed to complete common user web tasks across 10 of the most popular websites, including Google, Facebook, YouTube, Wikipedia, Twitter and Amazon [22]. That work also showed that this number only grew logarithmically with respect to the number of new web tasks added. The 11 web tasks we used in the study led to an average of 12.55 ($\sigma = 5.00$) user intents per task, including duplication. This suggests that our controlled experimental setting can closely represent the interaction patterns performed with larger-scale applications.

USE CASES

To use our techniques in the life cycle of software development, developers could embed a bug report UI on top of our proposed techniques to enable testers to submit the defects they encounter while testing. This is a similar approach from prior work [38] and what existing crowd testing platforms do in practice [45].

Automated testing tools like Selenium [1] rely on developers to come up with test scenarios, design and write test cases, and update them manually when a GUI changes, which can be tedious. By combining our techniques with these testing tools, developers can conduct (1) user behavior analysis by analyzing tester behavior, (2) combinatorial testing by combing different widgets across different user intents, and (3) integration testing by integrating each "unit" graph into the rest of the application graph, as in Fig. 7. This is because the outcome of our testers' output consists of an array of all the testers' actions in chronological order, which includes user actions (e.g., click, type), interacted DOM element ID, and other condition-related information, and a user intent graph that summarizes these testers' actions. The dense information of our output enables the design of diverse further use cases.

For example, a tester's output test case can be used as a real user behavior template that can guide combinatorial testing for detecting defects by interactions of parameters (i.e., GUI elements) across different user intents [20]. In our post survey, a tester also acknowledged this approach: "[testing the] same trace is not bad, because usually a lot of bugs appear when you check same trace with little bit different combination of data. I'm not always avoiding same traces. Just trying maximum combinations as possible" (P6). Furthermore, developers can run integration testing by integrating each "unit" graph into the larger application-level graph (Fig. 7) and test them together without recruiting more testers. This type of behavior-driven integration testing is possible because it builds on the reasonable assumption that integrating behavior-driven unit test cases would still be realistic if combined with test cases' shared GUI components.

LIMITATIONS

Our techniques can be most beneficial for testing GUIs with an object model (so that overlays can be drawn over specific elements) and a finite state space (so that the EFG can be rendered). This includes most web and mobile UIs. UIs with non-finite state spaces would need to collapse states together to make the graph readable to testers. For example, for a video playback widget where the user could scrub to an infinite number of playback positions, its states might be reduced to "start," "middle," and "end".

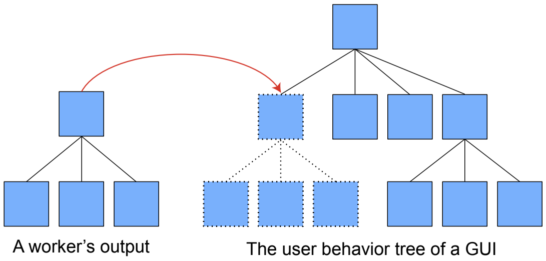


Figure 7. A small "unit" graph (left) can be integrated into larger application-level graph (right) to test the entire graph without recruiting additional testers.

FUTURE WORK

Our techniques rely on correctly identifying user intents (i.e., what is the user intent of each user action?) and Web structural semantics (i.e., what elements share the same user intent?). Prior work in the HCI community has explored methods to tackle these two challenging tasks [21, 17, 24]. Future work can (1) explore requesting the testers to annotate the possible user intents per element group and then use the aggregated annotations to instrument the GUIs, and (2) automate this process by predicting UI semantics and grouping them in the associated user intent [26]. Another future direction could be exploring the scalability of the interactive event-flow graphs, specifically, effectively guiding testers for high test coverage. While one could break down the large interactive event-flow graphs into smaller portions, it would be interesting to discover how to display a subset of nodes and edges that are more relevant to testers' current GUI state.

CONCLUSION

This paper proposes two new simple but effective GUI crowd testing techniques, interactive event-flow graphs and GUI-level guidance, to make the process more efficient. The interactive event-flow graphs track and aggregates testers' GUI actions in a directed graph that summarizes the navigation paths that have already been explored. This graph then provides GUI-level guidance directly in the form of overlay on the GUI that testers use, which helps them avoid creating duplicate test cases. Our results show that the guidance of these two techniques can effectively help both untrained and trained testers significantly increase their test coverage.

ACKNOWLEDGMENTS

We thank Rebecca Krosnick and Stephanie O'Keefe for their editing assistance, our anonymous reviewers for their helpful suggestions on this work, and our study participants for their time. This work was partially supported by Clinc, Inc.

REFERENCES

[1] 2019. Selenium browser automation. (2019). `https://www.seleniumhq.org/` Accessed: Sep, 2019.

[2] Amazon. 2018. Amazon Mechanical Turk. https://www.mturk.com/. Accessed: Sep, 2019.

[3] Applause. 2019. UTest. https://www.utest.com/. Accessed: Sep, 2019.

[4] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 571–580.

[5] Josh Attenberg, Panagiotis G Ipeirotis, and Foster J Provost. 2011. Beat the Machine: Challenging Workers to Find the Unknown Unknowns. *Human Computation* 11, 11 (2011), 2–7.

[6] Baidu. 2019. Baidu Crowd Test platform. http://test.baidu.com/crowdtest/crowdhome/guide. Accessed: Sep, 2019.

[7] Sebastian Bauersfeld and Tanja Vos. 2012. A reinforcement learning approach to automated gui robustness testing. In *Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012)*. 7–12.

[8] Penelope A Brooks and Atif M Memon. 2007. Automated GUI testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 333–342.

[9] Lydia B Chilton, Greg Little, Darren Edge, Daniel S Weld, and James A Landay. 2013. Cascade: Crowdsourcing taxonomy creation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1999–2008.

[10] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[11] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-objective crowd worker selection in crowdsourced testing. In *29th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 218–223.

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

[13] Biplab Deka, Zifeng Huang, Chad Franzen, Jeffrey Nichols, Yang Li, and Ranjitha Kumar. 2017b. ZIPT: Zero-Integration Performance Testing of Mobile App Designs. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 727–736.

[14] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1525–1534.

[15] Eelco Dolstra, Raynor Vliegendhart, and Johan Pouwelse. 2013. Crowdsourcing gui tests. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 332–341.

[16] Markus Ermuth and Michael Pradel. 2016. Monkey see, monkey do: effective generation of GUI tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 82–93.

[17] Forrest Huang, John F Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based User Interface Retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 104.

[18] Juha Itkonen and Mika V Mäntylä. 2014. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering* 19, 2 (2014), 303–342.

[19] Sean Kross and Philip J Guo. 2018. Students, systems, and interactions: synthesizing the first four years of learning@ scale and charting the future. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. ACM, 2.

[20] Rick Kuhn, Yu Lei, and Raghu Kacker. 2008. Practical combinatorial testing: Beyond pairwise. *It Professional* 10, 3 (2008), 19–23.

[21] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R Klemmer, and Jerry O Talton. 2013. Webzeitgeist: design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3083–3092.

[22] Walter Lasecki, Tessa Lau, Grant He, and Jeffrey Bigham. 2012. Crowd-based recognition of web interaction patterns. In *Adjunct proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 99–100.

[23] Walter S Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F Allen, and Jeffrey P Bigham. 2013. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 151–162.

[24] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.

[25] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining android app usages for generating actionable gui-based execution scenarios. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 111–122.

[26] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning design semantics for mobile apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, 569–579.

[27] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. 2015. A survey of the use of crowdsourcing in software engineering. *Rn* 15, 01 (2015).

[28] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 16–26.

[29] Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2013. Patina: Dynamic heatmaps for visualizing application usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3227–3236.

[30] Atif M Memon. 2002. GUI testing: Pitfalls and process. *Computer* 8 (2002), 87–88.

[31] Atif M Memon. 2007. An event-flow model of GUI-based applications for testing. *Software testing, verification and reliability* 17, 3 (2007), 137–157.

[32] Atif M Memon and Bao N Nguyen. 2010. Advances in automated model-based system testing of software applications with a GUI front-end. In *Advances in Computers*. Vol. 80. Elsevier, 121–162.

[33] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. 2001. Coverage criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 256–267.

[34] Yuan Miao and Xuebing Yang. 2010. An FSM based GUI test automation model. In *2010 11th International Conference on Control Automation Robotics & Vision*. IEEE, 120–126.

[35] Mark Micallef, Chris Porter, and Andrea Borg. 2016. Do exploratory testers need formal training? an investigation using hci techniques. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*. IEEE, 305–314.

[36] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 67–78.

[37] Brad A Myers. 1991. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST*, Vol. 91. Citeseer, 211–220.

[38] Michael Nebeling, Maximilian Speicher, Michael Grossniklaus, and Moira C Norrie. 2012. Crowdsourced web site evaluation with crowdstudy. In *International Conference on Web Engineering*. Springer, 494–497.

[39] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated software engineering* 21, 1 (2014), 65–105.

[40] Peter Pirolli and Stuart Card. 1999. Information foraging. *Psychological review* 106, 4 (1999), 643.

[41] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, 36–42.

[42] Katharina Reinecke, Tom Yeh, Luke Miratrix, Rahmatri Mardiko, Yuechen Zhao, Jenny Liu, and Krzysztof Z Gajos. 2013. Predicting users' first impressions of website aesthetics with a quantification of perceived visual complexity and colorfulness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2049–2058.

[43] Brian Robinson, Patrick Francis, and Fredrik Ekdahl. 2008. A defect-driven process for software quality improvement. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 333–335.

[44] Urko Rueda, Anna Esparcia-Alcázar, and Tanja EJ Vos. 2016. Visualization of automated test results obtained by the TESTAR tool.. In *CIbSE*. 53–66.

[45] Inc. UserTesting. 2019. UserTesting. https://www.usertesting.com/. Accessed: Sep, 2019.

[46] Luis Von Ahn and Laura Dabbish. 2004. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 319–326.

[47] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. 2015. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)* 6, 3 (2015), 46–83.

[48] Junjie Wang, Qiang Cui, Song Wang, and Qing Wang. 2017. Domain adaptation for test report classification in crowdsourced testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 83–92.

[49] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2018. Cutting Away the Confusion From Crowdtesting. *arXiv preprint arXiv:1805.02763* (2018).

[50] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 190–201.

[51] James A Whittaker. 2009. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education.

[52] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. 2007. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1129–1136.

[53] Miao Xie, Qing Wang, Guowei Yang, and Mingshu Li. 2017. Cocoon: Crowdsourced testing quality maximization under context coverage constraint. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE, 316–327.

[54] Qing Xie and Atif M Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, 1 (2007), 4.

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