Unfolding State Changes via Live State-First Debugging

Ruanqianqian (Lisa) Huang
r6huang@ucsd.edu
UC San Diego

Philip J. Guo
pg@ucsd.edu
UC San Diego

Sorin Lerner
lerner@cs.ucsd.edu
UC San Diego

ABSTRACT

Common debugging techniques are *execution-first*, requiring programmers to probe into execution via print logging or breakpoints to inspect intermediate program states. To alleviate the tedium of execution probing, *state-first* debugging techniques reveal state changes without requiring logs or statement-level breakpoints. Both techniques, however, remain time-consuming and laborious due to the need to manually sift through log or debugger outputs, and even more so when the process must be repeated many times due to code edits. To overcome these limitations, we propose *live state-first debugging*, a live programming paradigm that directly shows programmers where their program state has changed and how those state changes relate to code, all without requiring any logging or breakpoints. We implemented this paradigm for web-based GUI applications in Unfold, which shows a timeline of changed UI states, the corresponding code that caused those changes, and automatic replays of prior user interaction traces after the code edits are saved. A preliminary user study (N=12) shows that live state-first debugging helps programmers locate some GUI application bugs faster, and that programmers deem the paradigm usable and helpful.

KEYWORDS

debugging, GUI applications, live programming, event handling

1 INTRODUCTION

Debugging can be time-consuming and frustrating for programmers [24, 39, 44]. The most common debugging techniques are logging (i.e., inserting print statements) and single-step debuggers [32]. Both techniques allow programmers to inspect hidden intermediate states of their program's run-time data. However, their fundamental limitation is that they are *execution-first*: to see these intermediate states, programmers have to (1) probe into execution by manually inserting print statements or debugger breakpoints and (2) manually compare the observed intermediate states to figure out which ones diverge from their expectations and why. In addition, print statements or breakpoints must be *inserted at exactly the right places* to reveal the critical state change that led to the bug, which could be easily missed when too few or too many states are observed.

In contrast to the execution-first approaches, *state-first* techniques such as object-centric debugging [12, 27, 34, 35] and data breakpoints [43] allow programmers to navigate state changes without repetitive execution probing. While state-first techniques help programmers navigate state changes, they are still interruptive in nature: programmers see no connections from the state changes to the code, are often limited to examining one state change at a time, and need to repeat navigating to the state change of interest upon code changes. As debugging is an iterative activity with frequent edit-run cycles [5], these interruptions slow down this workflow.

To address this limitation, we extend state-first debugging with *liveness*. Live programming [17, 41] alleviates the need for frequent edit-run cycles by continuously providing up-to-date values, typically in the form of runtime values [25, 42] or traces [26] on each program change. As such, we propose *live state-first debugging*, a novel paradigm that integrates liveness with state-first debugging to live-update where execution state has changed and how *all* state changes relate to code. Adding liveness to state-first debugging is a technical and interaction design challenge due to the need to (a) constantly record and replay program inputs on the new code to obtain up-to-date state-first information and (b) support frequent context switches between editor and debugger. Beyond liveness, our paradigm also provides several improvements over prior state-first systems, namely: (1) visualizing state-to-code correspondences in situ, and (2) presenting all execution steps causing state changes and their stack traces at once via such correspondences.

As a proof-of-concept of live state-first debugging, we created a web-based GUI application debugger Unfold, which shows a timeline of changed UI states, what code caused those changes, and automatic replays of prior user interaction traces upon each code edit. We chose to implement live state-first debugging for GUI applications because these often involve difficult-to-debug state changes due to asynchronous event handling and user interactions. By always showing changed UI states and what code caused those changes, Unfold eliminates the need to repeatedly probe, pause, or restart execution or provide user interaction inputs.

Example usage scenario. Alice is a front-end web engineer trying to debug some faulty code that her colleague wrote. Her team is building a landing page for a shopping website. When the user clicks on a particular product description, the web app pops up a modal dialog box to show a promotion for it, and that pop-up should disappear when they hit the "close" button. However, there is a bug somewhere as the pop-up does not disappear when the "close" button is pressed. Fig. 1 shows how Alice can find and fix this bug using Unfold by interacting normally with the webpage without repeating her interactions or probing/restarting the execution.

(A) She clicks on a product on the webpage like a user does.
(B) Immediately she sees a pop-up of the promotion as expected. The UI States Timeline at the bottom of Unfold shows the initial page UI state labeled with a (0) circle, then two intermediate UI states labeled with (1) and (2) showing the pop-up appearing (1) and the page background darkening (2). This looks fine to her, so she clicks the "close" button.
(C) On the page the pop-up did not disappear as expected, which indicates a bug. But in the UI States Timeline, states labeled (3) and (4) show that the pop-up did disappear and is no longer visible in the UI, which seems strange to Alice.
(D) Inspecting further, when she scrolls to the right in the UI States Timeline, it shows that the pop-up did indeed disappear but *later came back* in the subsequent states labeled (5) and (6). By examining the code annotated with labels shared by the unwanted UI states (5) and (6), she realizes that the

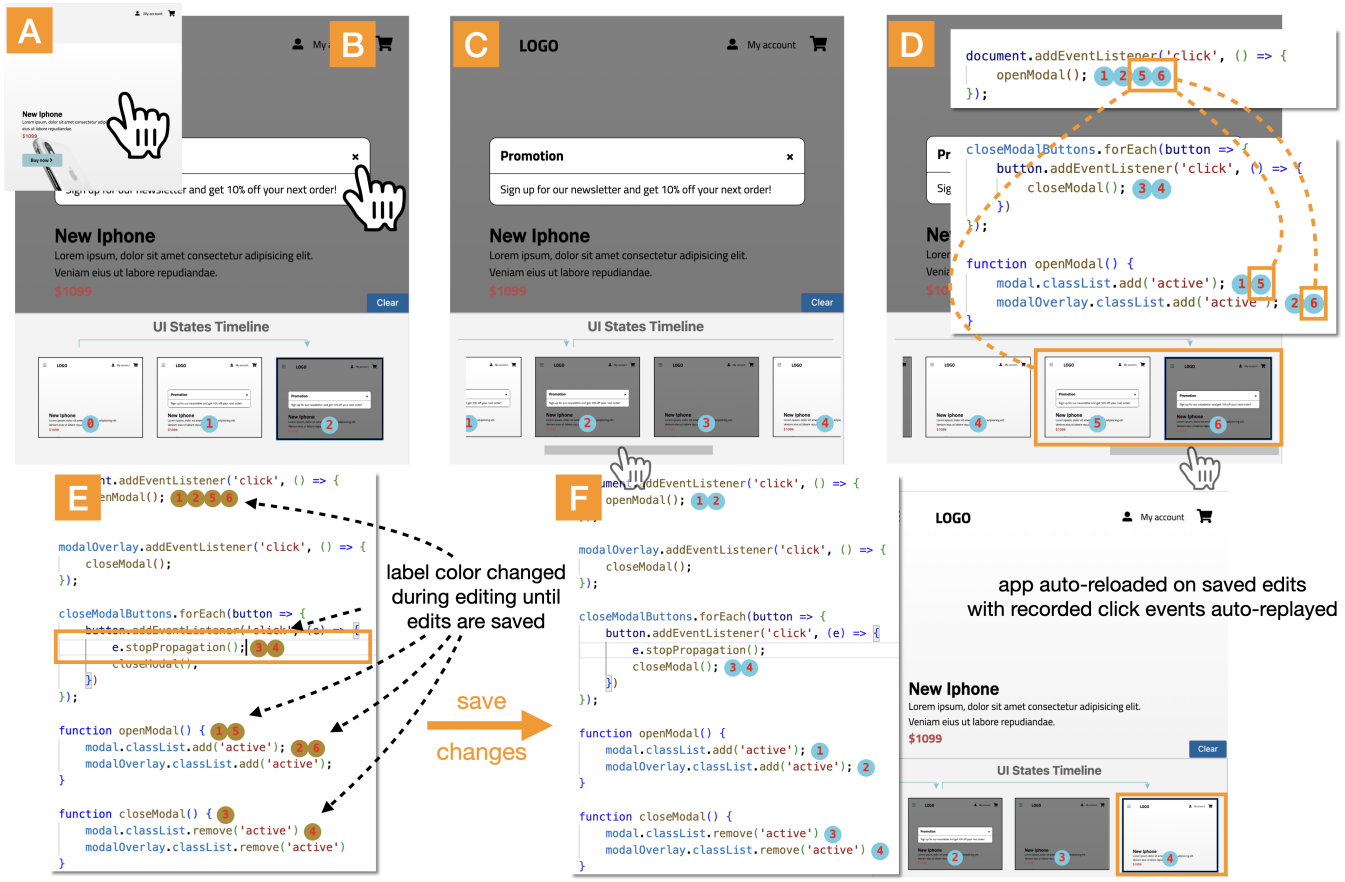

Figure 1: Unfold enables users to do *live state-first debugging* by interacting normally with the app, examining the timeline of UI states, seeing how buggy UI states connect to code, editing and saving the code, and immediately seeing updated UI states.

"click" event handler that shows the pop-up modal dialog box was called at a time when she did not expect it to be.

(E) Re-inspecting the structure of the DOM, Alice sees that this is a bug related to event bubbling [3]. The `click` event received on the "close" button bubbled up to the entire page (a parent DOM node), causing its own `click` event handler to execute. Alice quickly fixes this bug by adding a `stopPropagation()` call to the beginning of the `click` event handler for the "close" button to stop event bubbling.

(F) As soon as Alice saves her changes, Unfold live re-runs her code with the user interactions from her prior session, and now the pop-up disappears as she expects. She further confirms from the auto-refreshed UI states visualization that the application now behaves exactly as she expects.

Without Unfold, Alice would have needed to insert print logging or breakpoints, edit her code, reload the page, click to interact with it, inspect the logging or debugger output (which may contain too little or too much output), and iterate until she finds the bug. She might put logging or breakpoints in the wrong places at first and then need to re-run and repeat her page interactions a few times. A single button click may not seem so tedious, but a realistic debugging scenario involves a much longer chain of complex user interactions that need to be manually repeated on each trial. Unfold eliminates this tedium by showing UI state changes and live re-running the edited code with user interactions from prior runs.

We demonstrate the preliminary effectiveness of live state-first debugging through a comparative within-subjects study where 12 programmers debugged event handlers in GUI applications using Unfold versus standard browser developer tools. We found that programmers located some GUI application bugs faster with live state-first debugging and deemed the paradigm usable and helpful.

This paper's contributions to live programming research are:

- Live state-first debugging, a new paradigm that shows state changes and connections to the code that caused those changes
- A prototype of this paradigm in Unfold and a preliminary user study showing its potential effectiveness in locating GUI bugs.

2 RELATED WORK

State-First Debugging. We define state-first debuggers as systems where users see state changes without probing into the execution via logs or statement-level breakpoints. One example of state-first debugging are data breakpoints [43], which pause execution when data in the specified memory location changes. Object-centric debugging [12, 34, 35] extends data breakpoints by pausing execution

on changes of object instances. Flow-centric debugging [27] further extends object-centric debugging, enabling a view of program execution that bridges the control- and the object-flow.

Our live state-first debugging approach differs from traditional state-first debugging in four ways: (1) it enables live feedback on code edits, which state-first systems to date lack; (2) it provides a timeline of state changes, while the notion of an execution timeline is absent in prior work; (3) it shows state-to-code connections in source code, which visualizes control flow; (4) it supports back-in-time debugging on the state level, whereas prior work only provides such support on the execution level [27] or limits traveling back to specific object events only (e.g., instantiation) [35].

Live Programming. Live programming allows programmers to modify a program while receiving immediate feedback on the edits [40–42], typically in the form of runtime values [25, 42] or traces [26]. It has been applied to settings such as general-purpose languages [25, 31, 33], data science [13, 14, 45] , and physical computing [9, 10, 38]. Literature has found potential benefits of liveness for program comprehension [10, 13] and debugging [20, 23, 46]. However, rather than just showing changes in the execution state, prior live programming systems display runtime information for *one* or *all* steps of the execution and impose potential information overload and distraction on the user [22, 25]. While our work aims to enhance state-first debugging with liveness, we consider the state-first visualization a possible remedy to the problem of information overload [25] in live programming as well. Leveraging the benefits of both live programming and state-first debugging, our work explores how both approaches complement each other and how live state-first debugging supports locating and fixing bugs.

Debugging Tools for GUI Applications. Although live state-first debugging is a generic debugging paradigm, we implemented our first prototype Unfold for GUI applications. There is a rich history of debugging systems targetting GUI applications. Amulet [29] is among the first steps towards usable debugging tools for GUI apps, which supports changing data values at runtime. Timelapse explains changes of a specified DOM element [8], which, similar to Unfold, automatically captures user-interactions and replays execution. Clematis captures application behavior to a timeline of episodes and shows the event trace, execution trace, and DOM mutations for each [6]. Doppio visualizes high-level correspondences between callback methods and GUI changes in Android apps [11]. RDE [36] and Lively4 [28] support debugging GUI apps with liveness. However, none of these tools provide direct connections from UI states back to code upon edits, which Unfold implements.

3 FORMATIVE STUDY AND DESIGN GOALS

To establish design goals for Unfold, we ran a formative observational study with five programmers debugging JavaScript web event handling code similar to our example from Fig. 1. Specifically, we had them debug broken versions of a simple todo list app and a web-based spreadsheet, both of which involve synchronizing UI state changes with an underlying data model. Here we briefly summarize our observations and the three design goals for Unfold (D1, D2, D3) that we derived from this study, which can alleviate problems our participants encountered when using existing tools.

D1: Non-Interruptive Tracing of UI State Changes. Four out of our five participants chose not to use the (execution-first) debugger that comes with browser devtools [1, 15, 21] to inspect the web app's behavior because, as one said, "it is interruptive." They opted for print statements instead because execution-first debuggers prevent users from seeing a bigger picture of the execution without breakpoint pauses. The problem is exacerbated in GUI apps as debuggers interrupt tracing the UI states caused by event handlers, which to our participants was the most important. Execution-first debuggers force users to step through long, unrelated UI state changes with unavoidable pauses, which is interruptive and, as one participant stated, "does not simulate how the application naturally responds to a triggered event." Thus, our first design goal is non-interruptive tracing of state changes, which liveness precisely supports.

D2: Connecting Output Changes to the Corresponding Code. All five participants inserted print statements via `console.log()` to verify the execution of event handlers and determine the sequence of code execution, data changes, and UI state changes. However, they had to switch among the UI, code, and console to obtain such information. Console logs also could not immediately show the order of code execution and data changes unless carefully constructed and formatted. Participants wished to see the connection between code and changes in data and GUI output, especially the *order* of these changes, but they currently need to rely on manually sifting through textual logs to obtain such information indirectly.

D3: Automated Event Triggering. All participants found it tedious to manually trigger an event sequence repeatedly (e.g., via mouse clicks) when debugging event handlers, especially when the debugging took numerous rounds of editing. Also, changing one segment of event handling code might unexpectedly affect the behavior of another segment. When editing code for event handlers, users yearned for a way to "preview the effect of code changes on the execution of these handlers before committing to such changes." Live programming would free users from repeating event inputs and restarting the execution via its immediate feedback on code edits.

4 DESIGN & IMPLEMENTATION OF UNFOLD

We implemented a prototype of live state-first debugging for web-based GUI applications in a tool called Unfold, which is an extension to Visual Studio Code [4] that displays the user's code, an iframe containing the live webpage rendered from that code, and a UI States Timeline. We revisit the components of Fig. 1 and three design goals (D1, D2, D3) in Sec. 3 to describe Unfold.

Live Feedback on Code Edits. Unfold provides immediate feedback on how a code change affects the application's behavior whenever that change is saved, which supports both ***D1: Non-interruptive Tracing of UI State Changes*** and ***D3: Automated Event Triggering***. Liveness is enabled via event recording. Indeed, the user can interact with their web app's UI in the iframe through mouse clicks as usual, and Unfold will automatically record these clicks and replay them each time the code is changed and saved. Unfold records and replays one event sequence at a time to visualize the UI states created by that sequence, which is displayed in the UI States Timeline (next section). To input a new sequence, the user clicks the "Clear" button above the UI States Timeline (Fig. 2).

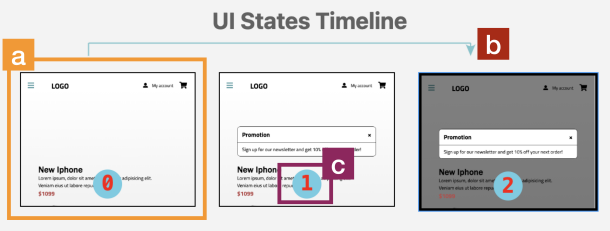

Figure 2: The UI States Timeline shows all UI states created by each recorded click event. (a) Each state can be expanded to see details; (b) an arrow connects the starting and ending states caused by this click event; (c) shows an intermediate UI state with a label (1), which also appears in the IDE over the line of code that caused this state change (see Fig. 1).

Our prototype records only `click` events, but its record-replay architecture can be extended to other DOM events due to our implementation. Specifically, when an event is captured, Unfold calculates the CSS selector of its event target and sets one breakpoint for that event via the Chrome debugger protocol [15]. It then simulates the event on the target via a headless Chrome instance and records related runtime information (next section).

UI States Timeline. Fig. 2 zooms in on the UI States Timeline from Fig. 1-A, which implements ***D1***. Unfold records snapshots of UI states by pausing execution at each click handler and single-stepping with the Chrome debugger protocol to take a full UI screenshot of the webpage at each executed line. Then it diffs consecutive screenshots and discards duplicates (which indicates that this line of code did not change the UI in a visible way). At the end of the click handler, all distinct screenshots appear as separate UI states created by running that handler. If a line of code runs an animation (e.g., fade-in), then it will record the UI state at the start and end of that animation. Note that if the click handler runs many lines of code, then this approach can become slow and we have not yet optimized for performance; but in practice, developers often write click handlers to be short-running so that the UI updates quickly to foster interactivity [30].

Above the sequence of UI states in Fig. 2, an arrow (Fig. 2-b) is shown for *each* `click` event – the arrow spans all UI states that the UI passes through when that event is handled. In this example it shows the starting UI state of the webpage (labeled with a (0)), the pop-up modal dialog appearing (state (1)), and then the background of the page darkening (state (2)).

Our formative study participants (Sec. 3) all found print statements to be useful. Thus, in addition to inspecting UI states in this timeline, Unfold allows the user to also use console print logging for inspecting internal program state. But instead of logging to a separate console, Unfold embeds the logs in the GUI so that each log output results in a visible UI state change, thus enabling users to examine log output directly within the UI States Timeline. This mechanism enables users to reason about the sequence of internal data changes along with visible UI changes together in one timeline.

UI-to-Code Connector Labels. The numbered circles shown on UI states (Fig. 2-c) visually connect those UI states to code execution, which addresses ***D2: Connecting Output Changes to the Corresponding Code***. The semantics of these numbered labels is

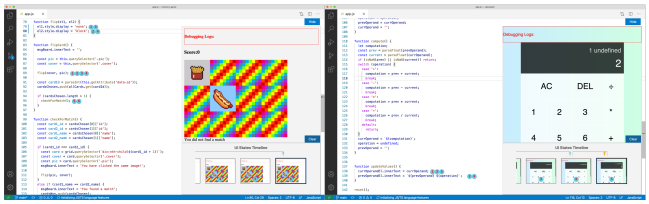

Figure 3: Applications in our user study, displayed in Unfold: Memory Game (left) and Calculator (right). Each has ~90 lines of JavaScript code and contains two bugs we inserted.

defined as follows: Suppose we have a UI state annotated with a label i, and a line of code annotated with the same label. This means two things: (1) right before the i-labeled line of code started executing, the state of the UI was the one annotated with $i-1$ in the UI States Timeline; (2) right after the i-labeled line of code finished executing, the state of the UI was the one annotated with i in the timeline. These labels use numerical ordering to visualize control flow in the source code at a glance and connect them to UI state changes. Fig. 1 shows more examples of labels next to lines of code.

5 EVALUATION: COMPARATIVE USER STUDY

State-first debugging aims to alleviate the tedium of execution probing in execution-first debugging, but execution-first debugging remains the mainstream debugging paradigm as the former fails to address interruptions in debugging. With liveness, can state-first information help programmers locate and fix bugs in GUI apps? How does live state-first debugging compare to execution-first debugging? To investigate these questions, we ran a within-subjects study where each participant debugged two GUI applications using Unfold and Firefox DevTools [1] in a randomized order, and we collected task performance and user impressions of each tool.

Participants. We recruited 12 adults (6 female) with 1 to 15 years of JavaScript experience and 5 to 15 years of programming experience via personal contacts, social media, and our institution.

Tasks. Each participant debugged two GUI applications, Memory Game (M) and Calculator (C)[1], and Fig. 3 illustrates these applications in Unfold. Each comes with two bugs: one directly related to event-driven DOM/CSS manipulations, a critical part of JavaScript event handling [6], and the other caused by data changes. Here are the bugs by application (M and C) labeled with bug type:

- M1. [CSS] A card disappears when trying to flip it over.
- M2. [Data] Off-by-one error to flip back a pair of flipped cards.
- C1. [DOM] Pressing any value key shows `undefined`.
- C2. [Data] Pressing "=" after an operation makes no calculation.

Conditions. Participants edited code in the VSCode IDE [4] in the study with two different debugging setups on top: live state-first (Unfold) and execution-first (Firefox DevTools [1] Control condition). Given two setups and two applications, we randomized the order of the setups and the applications to reduce learning effects, so we had four groups. We randomly assigned the 12 participants to groups while maintaining even group sizes (three participants each).

[1]Modified from examples found in intermediate-level tutorials on JavaScript event handling: bit.ly/sfd-memory-game and bit.ly/sfd-calculator, respectively.

Procedure. We used video conferencing for the study. At the beginning of each task, we gave a 10-minute tutorial on the debugging setup using the same tutorial code and allowed participants to ask clarifying questions. They then had 30 minutes to locate and fix the two bugs in their desired order. They were allowed to search for documentation, encouraged to think aloud, and required to tell us when they located or fixed a bug. We stopped the timer for the application when they either asked to move on or used up all 30 minutes. After debugging two applications, we spent the final 15 minutes giving them a survey and a semi-structured interview.

Quantitative Data. We recorded screencast videos of each session and measured (1) duration and (2) success of both locating and fixing bugs by reviewing the recordings. We measured the time to locate a bug as the duration from the start of working on an app or the end of locating/fixing the previous bug until the participant said they found it or the time was up. Similarly, we measured the time of fixing a bug as the duration from the end of locating the bug, if any, until the participant said they fixed it or time was up. Finally, we marked that a participant succeeded in locating bugs in one app if both bugs were correctly located, and that they succeeded in fixing bugs if both bugs were correctly located and fixed.

Qualitative Data. We recorded participant quotes during the study and interview. We also obtained open-ended responses and Likert-scale ratings on features of Unfold from the survey we gave right after the tasks were done. One author coded responses using thematic analysis [7] and the other authors reviewed these codes.

Study Limitations. First, the study is limited in its sample size of 12 participants, who might not be representative enough, and whose self-reported programming experience might have biased the study results. Second, there were only two small GUI apps used in our study, and we only compared Unfold to one execution-first debugging tool (Firefox DevTools). Thus, we do not know how well this technique generalizes to larger, more complex web applications that run in production environments. As such, our study findings should be viewed as an early step towards evaluating the effects of live state-first debugging for GUI applications or implementations of the paradigm for other domains.

5.1 Effectiveness in Debugging

Duration of Locating Bugs. Fig. 4 shows the duration of locating bugs by application and debugging setup. Since participants were given 30 minutes to work on each application, including locating and fixing bugs, the maximum time one could spend locating bugs was 30 minutes. Using a Wilcoxon signed-rank test on median values, we found that participants with Unfold located bugs *significantly* faster in Calculator, by 15.7 minutes ($p = .036$), but only marginally faster in Memory Game ($p = .562$), by 0.9 minutes.

Duration of Fixing Bugs. Among all participants, we only compared the duration of those who eventually succeeded in fixing the bugs. Median values show that participants using Unfold spent 3.5 minutes more fixing bugs in Memory Game but 3 minutes less in Calculator than those with Control. The sample sizes were too small (all ≤ 3) to conduct statistical tests and conclude any effects.

Success in Locating Bugs. In Memory Game, five out of six participants located all bugs in Unfold and six did so in the Control condition. In Calculator, six participants located all bugs in Unfold and four out of six did so in Control. The differences in results are not statistically significant according to Fisher's exact test.

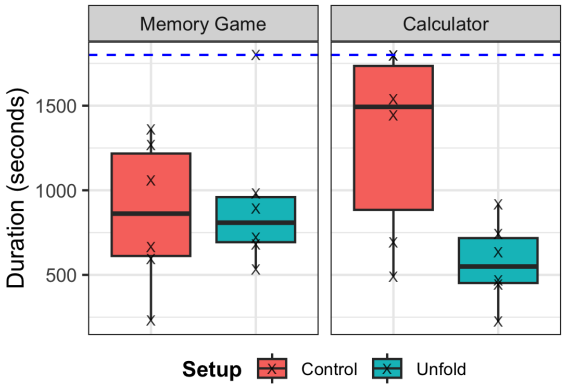

Figure 4: Duration of locating bugs by application and setup. The "X" marks represent data points. Two data points overlap at the upper left corner of the Calculator plot (right). The center lines show the median values. The dashed horizontal line represents the time limit (1800 seconds = 30 minutes).

Success in Fixing Bugs. In Memory Game, three out of six participants fixed all bugs in Unfold, and three out of six did so in the Control condition. In Calculator, three out of six participants fixed all bugs in Unfold, and two out of six did so in Control. Fisher's exact test did not show any significant difference caused by the debugging setup in the success of fixing all bugs.

> ***Takeaway 1**: Participants located bugs significantly faster with Unfold in one application. We observed no difference in the success of locating or fixing bugs and time spent fixing bugs.*

5.2 Users' Perceptions of Unfold

Unfold Reduces Barriers to Debugging GUI Applications. The left part of Table 1 shows the ratings on how easy it was to use/understand Unfold on a 5-point Likert scale. Participants generally found it easy to use/understand: all ratings have averages and medians above 4 out of 5 (Strongly Agree). They mentioned three particular aspects of Unfold that ease debugging GUI apps.

Straightforward Interaction Flow. Participants found the Unfold interface to be "very intuitive" (P2, P3). P1 especially liked "its simplicity," and P2 appreciated the flat learning curve.

Minimal Configuration Requirement. All participants found that Unfold was "easy to setup," required "[few] context shifts" (P8), and picked up their "existing [work practice] … without [requiring] configuring what events to record" (P2).

Minimal Distraction. Participants considered that features in Unfold did not distract them much from debugging. Although the UI-code connector labels could be distracting "when there [were] too many UI states" (P3, P9, P11), the "ease of resetting" addressed this problem by clearing the recorded events and states.

> ***Takeaway 2**: Participants found Unfold easy to use and understand through a straightforward interaction flow, minimal configuration requirement, and reduced information overload.*

Table 1: Likert-scale user perceptions of Unfold features from 1 - "Strongly Disagree" to 5 - "Strongly Agree". "Avg." - Average, "Mdn." - Median, "Dist." - Distribution.

	Easy to use/understand			Helpful for debugging		
Feature	Avg.	Mdn.	Dist.	Avg.	Mdn.	Dist.
Event Recording	4.67	5.0		4.83	5.0	
Event Arrows	4.67	5.0		4.25	4.5	
Connector Labels	4.25	4.5		4.33	4.0	
Logging	4.08	4.0		4.17	5.0	
Live Feedback	4.00	4.0		4.42	5.0	
Overall	4.42	5.0		4.50	4.5	

Unfold Helps Debug GUI Applications. The right part of Table 1 shows participants' ratings for Unfold's helpfulness, also on a 5-point Likert scale. Broadly speaking, participants found it helpful, especially "Event Recording" that received an average rating of 4.83 out of 5. Participants deemed Unfold helpful in two specific ways.

UI States-to-Code Connections. 10 out of 12 participants mentioned that Unfold helped them reason about the behavior of the invoked event handlers by visually showing UI states and connecting the states to code execution. P8 especially appreciated the ability to "quickly understand the effects of [their] actions."

Live Feedback via Event Recording. Eight participants said Unfold greatly helped them debug by automating recorded events and providing live feedback upon code changes using these events. P7 further commented that, when trying to understand the effects of edits, they were able to "quickly compare the difference in app behavior between the changes" through the live feedback.

> ***Takeaway 3****: Participants deemed Unfold helpful for debugging as it helps understand code execution through UI states-to-code connections and provides live feedback upon code changes.*

Suggestions for Unfold. Participants also suggested possible improvements to Unfold that allow them to: (1) inspect runtime data along with the UI states without logging (P2); (2) filter uninteresting UI states and connector labels (P3, P9, P10, P11); (3) examine UI components in detail in a UI state (P6); and (4) adjust the granularity of information shown (P11). The suggestions share a common theme: control over run-time state inspection.

> ***Takeaway 4:*** *Participants yearned for more control over run-time state inspection in Unfold.*

6 DISCUSSION AND FUTURE WORK

When a Debugger Shows *How* to Fix Bugs. In our study, P9 and P12 located the bugs in Memory Game using Unfold but did not know how to fix them. There are two possible causes: (1) they were unfamiliar with the API calls that caused the bugs, which could be addressed via documentation and code examples search but require more time and context switches; (2) they had an expected UI state in mind but did not know how to achieve it via code. Indeed, while live state-first debugging may help understand code behavior and locate bugs, it cannot suggest fixes. We feel that *program synthesis* is well-suited for suggesting API usage and code edits. For example, millions are using GitHub Copilot [2] to obtain code suggestions within their IDE. Particularly for reaching expected UI states, synthesis via direct manipulation can also help and has been explored [18, 19, 37]. Still, an open problem remains in *validating* the synthesized code against the programmer's expectation. Recent work has observed the potential benefits of liveness for exploring synthesized code [16]. Live state-first debugging, by showing correspondences between execution state changes and the code on top of liveness, can provide further help with understanding and refining the synthesized repair suggestions. *Future debuggers could help programmers not only locate but also fix bugs by integrating aspects of live state-first debugging and program synthesis.*

Human-Centered Liveness. Our system Unfold immediately displays the result of the program logic in situ as it combines liveness with the browser core functionality, allowing the programmer to focus on finding program misbehavior in GUI applications and thus debug faster in some tasks. We attribute the promising results to the use of human-centered design for live state-first debugging. When bringing liveness and state-first debugging to web-based GUI applications, we grounded the design in results from a formative study taking into account challenges and user needs in this domain. As AI programming assistants become increasingly popular, suggesting code within seconds, it is crucial for programming tools to provide *timely* feedback on the *correctness* of code suggestions. We deem live programming as one possible mechanism to deliver such timely feedback. As the live programming community prepares for tool advances that support AI-assisted programming, it is important to also embrace the diversity of programming tasks and domains. *We call for using a human-centered design that accounts for domain-specific user needs to develop future live programming systems that are truly effective.*

Usable Programming Systems. We developed live state-first debugging out of the need for combining liveness and state examination in debugging based on prior work and a formative study. Despite the promising results from the user evaluation, we have yet to understand why live state-first debugging did not help as much in certain debugging tasks. Looking back, while the community has been pushing the boundaries much further for programming systems — both live programming (e.g., [25, 26, 33, 42]) and state-first systems (e.g., [12, 27, 34]) — little research has addressed what makes these systems usable (or not). In the future, we plan to analyze the qualitative evidence of our user evaluation in more depth to seek improvements for live programming and state-first debugging. *As a community, we should build future programming systems based on systematic evaluations of existing ones in the same category.*

7 CONCLUSION

We proposed *live state-first debugging*, a liveness-enabled debugging paradigm that shows where the execution state has changed and how those state changes relate to code. We implemented the paradigm in Unfold for debugging GUI applications and found that live state-first debugging helped programmers locate some GUI bugs faster and was deemed usable and helpful. Our findings can inform opportunities for future live programming and debugging tools in terms of support for fixing bugs, design, and evaluation.

REFERENCES

[1] 2022. Firefox DevTools User Docs — Firefox Source Docs Documentation. https://firefox-source-docs.mozilla.org/devtools-user/.

[2] 2022. GitHub Copilot · Your AI Pair Programmer. https://github.com/features/copilot.

[3] 2023. Introduction to events - Learn web development. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

[4] 2023. Visual Studio Code - Code Editing. https://code.visualstudio.com/

[5] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576170

[6] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, Hyderabad India, 367–377. https://doi.org/10.1145/2568225.2568268

[7] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[8] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. ACM, St. Andrews Scotland, United Kingdom, 473–484. https://doi.org/10.1145/2501988.2502050

[9] Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. Programs in the Palm of Your Hand: How Live Programming Shapes Children's Interactions with Physical Computing Devices. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*. ACM, Boise, ID, USA, 227–236. https://doi.org/10.1145/3311927.3323138

[10] Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. Does live programming help program comprehension?–A user study with Live Robot Programming. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, Amsterdam, Netherlands, 8 pages. http://bergel.eu/MyPapers/Camp16-ComprehensionWithLRP.pdf

[11] Pei-Yu (Peggy) Chi, Sen-Po Hu, and Yang Li. 2018. Doppio: Tracking UI Flows and Code Changes for App Development. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–13. https://doi.org/10.1145/3173574.3174029

[12] Claudio Corrodi. 2016. Towards Efficient Object-Centric Debugging with Declarative Breakpoints.. In *SATToSE*. 32–39.

[13] Robert DeLine and Danyel Fisher. 2015. Supporting Exploratory Data Analysis with Live Programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[14] Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–11. https://doi.org/10.1145/3411764.3445267

[15] Chrome Developers. 2022. Chrome DevTools - Overview. https://developer.chrome.com/docs/devtools/overview/

[16] Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2023. Live Exploration of AI-Generated Programs. https://doi.org/10.48550/arXiv.2306.09541 arXiv:2306.09541 [cs].

[17] Christopher Michael Hancock. 2003. *Real-time programming and the big ideas of computational literacy*. Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/61549

[18] Brian Hempel and Ravi Chugh. 2022. *Maniposynth: Bimodal Tangible Functional Programming*. Technical Report. https://doi.org/10.4230/LIPIcs.ECOOP.2022.16 arXiv:2206.14992 [cs]

[19] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292. https://doi.org/10.1145/3332165.3347925 arXiv:1907.10699 [cs]

[20] Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 495–501. https://doi.org/10.1145/3478431.3499305

[21] Apple Inc. 2022. Tools - Safari. https://developer.apple.com/safari/tools/.

[22] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[23] J. Kramer, J. Kurz, T. Karrer, and J. Borchers. 2014. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013 ISSN: 1943-6106.

[24] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 383–392. https://doi.org/10.1109/ESEM.2013.43 ISSN: 1949-3789.

[25] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[26] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[27] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. 2009. Flow-Centric, Back-in-Time Debugging. In *Objects, Components, Models and Patterns (Lecture Notes in Business Information Processing)*, Manuel Oriol and Bertrand Meyer (Eds.). Springer, Berlin, Heidelberg, 272–288. https://doi.org/10.1007/978-3-642-02571-6_16

[28] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*. ACM, Vancouver BC Canada, 28–35. https://doi.org/10.1145/3167109

[29] Brad Myers, Alan Ferrency, Rich McDaniel, and Roger Dannenberg. 1996. Debugging Interactive Applications. (Jan. 1996). https://doi.org/10.1184/R1/6621842.v1

[30] J. Nielsen. 1993. Iterative user-interface design. *Computer* 26, 11 (Nov. 1993), 32–41. https://doi.org/10.1109/2.241424 Conference Name: Computer.

[31] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

[32] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (March 2017), 83–110. https://doi.org/10.1007/s11219-015-9294-2

[33] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. https://doi.org/10.22152/programming-journal.org/2019/3/9

[34] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. 485–495. https://doi.org/10.1109/ICSE.2012.6227167 ISSN: 1558-1225.

[35] Maximilian Ignacio Willembrinck Santander, Steven Costiou, Adrien Vanègue, and Anne Etien. 2022. Towards Object-centric Time-traveling Debuggers. (2022).

[36] Christopher Schuster and Cormac Flanagan. 2015. Live Programming for Event-Based Languages. In *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, Vol. 15. https://users.soe.ucsc.edu/~cormac/papers/15rebls.pdf

[37] Christopher Schuster and Cormac Flanagan. 2016. Live programming by example: using direct manipulation for live program synthesis. In *LIVE Workshop*. https://chris-schuster.net/live16/live16-lpbe.pdf

[38] Emmanuel Senft, Michael Hagenow, Robert Radwin, Michael Zinn, Michael Gleicher, and Bilge Mutlu. 2021. Situated Live Programming for Human-Robot Collaboration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 613–625. https://doi.org/10.1145/3472749.3474773

[39] Diomidis Spinellis. 2018. Modern debugging: the art of finding a needle in a haystack. *Commun. ACM* 61, 11 (Oct. 2018), 124–134. https://doi.org/10.1145/3186278

[40] Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages & Computing* 1, 2 (June 1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[41] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[42] Bret Victor. 2012. Learnable Programming. http://worrydream.com/LearnableProgramming/.

[43] Robert Wahbe, Steven Lucco, and Susan L Graham. 1993. Practical data breakpoints: Design and implementation. *ACM SIGPLAN Notices* 28, 6 (1993), 1–12.

[44] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368 Conference Name: IEEE Transactions on Software Engineering.

[45] Xiong Zhang and Philip J. Guo. 2017. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, Québec City QC Canada, 691–702. https://doi.org/10.1145/3126594.

3126663

[46] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces (IUI '22)*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120