Explaining Visual Changes in Web Interfaces

Brian Burg,[1] **Andrew J. Ko,**[1,2] **Michael D. Ernst**[1]

Computer Science and Engineering[1]
University of Washington
–burg, mernst˝@cs.washington.edu

The Information School[2]
University of Washington
ajko@uw.edu

ABSTRACT

Web developers often want to repurpose interactive behaviors from third-party web pages, but struggle to locate the specific source code that implements the behavior. This task is challenging because developers must find and connect all of the non-local interactions between event-based JavaScript code, declarative CSS styles, and web page content that combine to express the behavior.

The Scry tool embodies a new approach to locating the code that implements interactive behaviors. A developer selects a page element; whenever the element changes, Scry captures the rendering engine's inputs (DOM, CSS) and outputs (screenshot) for the element. For any two captured element states, Scry can compute how the states differ and which lines of JavaScript code were responsible. Using Scry, a developer can locate an interactive behavior's implementation by picking two output states; Scry indicates the JavaScript code directly responsible for their differences.

Author Keywords

Programming, debugging, web development, reverse engineering

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces

General Terms

Human Factors; Design

INTRODUCTION

Web developers increasingly look to existing web site designs for inspiration [12], to learn about new practices or APIs [5], and to copy and adapt interactive behaviors for their own purposes [28]. Web sites are particularly conducive to reuse because web pages are widely available, distributed in source form, and inspectable using tools built into web browsers.

Unfortunately, when a developer finds an interactive behavior that they want to reuse (e.g., a nicely designed widget, a parallax effect, or a slick new scrolling animation), finding the code that implements the behavior code from a third-party web site is still a challenging process [17]. Locating this code typically involves at least three tasks. First, a developer identifies a behavior's rendered *outputs* and speculates about the internal states that produce them; second, she observes changes to these outputs over time to find their *internal states*; third, she searches the web site's client side implementation to find the JavaScript *source code* that implements the behavior.

Even if all of these tasks go well, extracting the implementation of the desired behavior might require repeated experimentation and inspection of the running site. More often, the web's combination of declarative CSS, imperative JavaScript, and notoriously complex layout/rendering algorithms makes the feature location task prohibitively difficult. Moreover, as the web sites and web applications become more powerful, they have also become more complex, more obfuscated, and more abstract, aggravating these challenges.

While prior work has investigated feature location techniques for statically-typed, non-dynamic languages, no prior work comprehensively addresses the specific difficulties posed by the web. (1) Isolating the output, internal state and source code for single widget on a web page is difficult due to hidden and non-local interactions between the DOM, imperative JavaScript code, and declarative CSS styles. Some prior work has addressed this by revealing *all* hidden interactions [24], but this obfuscates critical example-relevant details in a flood of low-level information. (2) Web developers can view program states and outputs over time by repurposing tools for logging, profiling, testing, or deterministic replay [20, 2, 6], but none of these tools is designed to compare past states, and they cannot limit data collection to specific interface elements or behaviors. (3) No prior work exists that can attribute changes in web page states to specific lines of JavaScript code. Instead, web developers often resort to using breakpoints, logging, or browsing program text in the hope of finding relevant code [6, 18]. In other programming environments, research prototypes with this functionality all incur run-time overheads that limit their use to post-mortem debugging [14, 16, 23].

To address these gaps, we present Scry, a reverse-engineering tool that enables a web developer to (1) identify visual states in a live execution, (2) browse and compare relevant program states, and (3) jump directly from state differences to the JavaScript code responsible for the change. (1) To locate an interactive behavior's implementation using Scry, a developer first identifies a web page element to track. Whenever the selected interface element is re-drawn differently, Scry automatically captures a snapshot of the element's visual appearance and all relevant internal state used to render it. (2) Scry presents an interactive diff interface to show the CSS and DOM differences that caused any two visual states to differ, overlaying inline annotations to compactly summarize CSS and DOM changes between two visual states. (3) When a developer clicks an annotation, Scry reveals the operations that caused the output change and the corresponding JavaScript code that performed the operation. Scry supports these capabilities by capturing state snapshots, logging a trace of relevant mutation operations, and tracking dependencies between operations.

Figure 1. A picture mosaic widget that periodically switches image tiles with a cross-fade animation. It is a jQuery widget implemented in 975 lines of uncommented, minified JavaScript across 4 files. Its output is produced using the DOM, CSS animations, and asynchronous timers.

The result is a powerful, direct-manipulation, before-and-after approach to feature location for the web, eliminating the need for developers to speculate about and search for relevant code.

This paper begins with an illustration of how Scry helps a developer as they locate the code that implements a mosaic widget. Then, it explains the design rationale and features of Scry's user interface, and it describes Scry's snapshotting, comparison, and dependency tracking techniques. It concludes with several real-world case studies, related work, and future directions.

EXAMPLE

To illustrate Scry in use, consider Susan, a contract web developer who is overhauling a non-profit organization's website to be more engaging and interactive. While browsing another website[1], she finds a compelling picture mosaic widget (Figure 1) that might work well on the non-profit's donors page. To evaluate the widget's technical suitability, she needs a high-level understanding of how it is implemented in terms of DOM and CSS manipulations and the underlying JavaScript code. In particular, she wants to know more about the widget's cross-fade animation: its dependencies on specific DOM and CSS features, its configurability, and ease of maintenance. At this point, Susan is only superficially familiar with the example: how it looks visually and a vague intuition for what it does operationally. She is unfamiliar with the example's source code, and she does not desire a complete understanding of it unless absolutely necessary.

Existing developer tools provide several approaches for Susan to reverse-engineer the mosaic widget to gain this understanding, but all of these are ill-suited for her task. She could search through the page's thousands of lines of source code for functions and event handlers relevant to the mosaic and then try to comprehend them. Susan is unlikely to pursue this option as it is extremely time-consuming, and it might not aid her evaluation. She could inspect the page's output to see its related DOM tree elements and active CSS rules, but this only shows the page's current state and does not explain how the page's DOM tree or styles were constructed. She could use source-level tools (e.g., execution profiler, logging, breakpoints) to see what code actually executes when the widget animates. However, the efficient use of these tools requires *a priori* awareness of what code is relevant, and Susan is unfamiliar with the example's code.

[1] `https://www.mozilla.org/en-US/mission/`

Using Scry to Reverse-Engineer the Mosaic

Using Scry, Susan can identify the mosaic's relevant visual outputs, compare internal states that produced each output, and jump from internal state changes to the responsible JavaScript code. Instead of guessing about program states and searching static code, Susan's workflow is grounded by output examples, captured DOM and CSS states, and specific lines of JavaScript code. Susan starts by finding the mosaic's corresponding DOM element using the Web Inspector, and then tells Scry to track changes to the element (Figure 2.a). As mosaic tiles update, Scry captures snapshots of the mosaic's internal state and visual output. After several tile transitions, Susan stops tracking and browses the collected snapshots to identify visual states before, during, and after a single tile changes images (Figure 2.b).

Susan now wants to compare these output examples to see how their internal states differ as the cross-fade effect progresses. To do so, she selects two screenshots from the timeline (Figure 2.b). For each selected screenshot, Scry shows the small subset of the page's DOM (Figure 2.b) and CSS (Figure 2.c) that determines the mosaic's visual appearance at that time. By viewing the specific inputs and outputs of the browser's rendering algorithm at each instant, Susan can figure out how the mosaic widget is structured and laid out using DOM elements and CSS styles. To highlight dynamic behaviors, Scry visualizes differences between the states' DOM trees and CSS styles (Figure 2.c). Seeing that the tiles' `background-image` and `opacity` properties have changed, Susan now knows which CSS and DOM properties the mosaic uses to implement the cross-fade.

To find the code that causes these changes, Susan compares the DOM trees of the initial state and mid-transition state, noticing that the new tile initially appears underneath the original tile, and the original tile's `opacity` style property differs between the two states. When she selects the new tile's DOM element, Scry displays a list of JavaScript-initiated mutation operations that created and appended DOM elements for the new tile (Figure 2.d). To see how the tile's `opacity` property is animated, she clicks on its diff annotation and sees JavaScript stack traces for the state mutations that animate the style property (Figure 2.e,f). Susan now knows exactly how the widget's JavaScript, DOM, and CSS code works together to animate a tile's cross-fade transition. If Susan wants to modify the animation code, she now knows several places within the code from which to expand her understanding of the program.

A STAGED INTERFACE FOR FEATURE LOCATION

User interfaces for searching and understanding code can quickly become overwhelming, displaying large amounts of source code to filter, browse, and comprehend [25]. Scry's interface simplifies this work by identifying and supporting three distinct activities through dedicated interfaces: (1) the

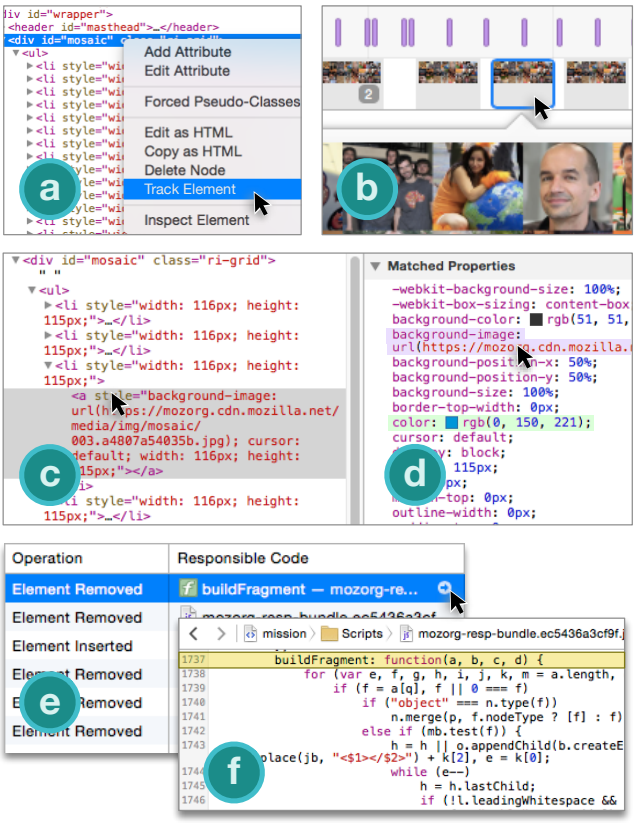

Figure 2. Susan first uses the Web Inspector to go from the mosaic's visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual states to inspect (b), and see the DOM tree (c) and CSS styles (d) that produced each visual state. To jump to the code that implements interactive behaviors, Susan uses Scry to compare two states and then selects a single style property difference (d). Scry shows the mutation operations responsible for causing the property difference (e), and Susan can jump to JavaScript code (f) that performed each mutation operation.

user identifies the behavior's major visual states, (2) she builds a mental model of how visual outputs are related to internal states, and (3) she explores how multiple internal states are connected via scripted behaviors. This section first describes and justifies this output- and difference-centric workflow; then, it explains how Scry's design supports a web developer during each of these feature-location activities.

Design Rationale

We designed Scry to directly address the information overload a developer encounters when performing feature location tasks [25]. Scry's design differs from traditional feature location tools in two fundamental ways: (1) Scry represents program states by their visual output whenever possible, and (2) Scry promotes a staged approach to feature location by iteratively showing more detailed information. We explain each of these points below.

Representing Interface States as Screenshots

Scry's user interface removes much of the guesswork from feature location, by using visual outputs as the primary basis for identifying and comparing an interactive behavior's intermediate internal states. Scry shows multiple output examples for an element along with the internal states (CSS styles and DOM elements) used to render each output (Figure 3). A user browses internal states by selecting the corresponding screenshots that each internal state produces. This output-based, example-first design is in contrast to the traditional tooling emphasis on static, textual program representations. During feature location tasks, browsing program states via output examples is a better match for what the user knows (a visual memory of a page's output) and what they lack but are seeking (knowledge of relevant state and code). Output examples are also more readily available: visual states are easier for developers to recognize and compare than internal states or static code, and output often changes in response to distinct and memorable user actions.

Performing Feature Location in Stages

Scry's interface allows a developer to pursue specific feature location tasks in relative isolation from each other. When a user wants to identify outputs, relate outputs to internal states, or connect state changes to source code, Scry provides only the information appropriate to each task.

To see the internal states that produced a single visual output, they can do so with without considering scripted behaviors and other interactions. While most interactive behaviors are scripted with JavaScript[2], ultimately an element's visual appearance is solely determined by its CSS styles and a DOM tree. Thus, to see how one visual state is produced, it is sufficient to understand the CSS and DOM that were used to render it. Scry supports this task by juxtaposing each screenshot with its corresponding DOM tree and computed CSS style properties (Figure 3).

To direct a user towards the internal states (CSS and DOM) responsible for visual changes, Scry's interface visualizes differences between two screenshots and their corresponding DOM and CSS states (Figure 4). Sometimes, inspecting the internal state and visual output of single visual state is insufficient for a useful mental model of how internal inputs affected visual output. If the user has a weak understanding of CSS or layout algorithms, or if the interface element is excessively large or complex, then it may be difficult to localize a visual effect to specific CSS styles and DOM elements. By juxtaposing small changes in internal state with the corresponding visual outputs, Scry prompts a user to test and refine their mental model against a small, understandable example. These diffs also reveal the means by which JavaScript code is able to transition between different visual states.

To help a user understand how state differences came to be and what code was responsible, Scry explains how each DOM and CSS difference came to be in terms of abstract *mutation operations* that modify CSS styles or the DOM tree. Each mutation operation serves a dual purpose: it jointly explains how internal state changed, and also provides a starting point

[2]Simple interactions can be programmed entirely within declarative style rules using CSS animations, transitions, and pseudo-states (i.e, `:hover`, `:focus`) to specify keyframes. Scry can track these internal state changes even though no JavaScript is involved.

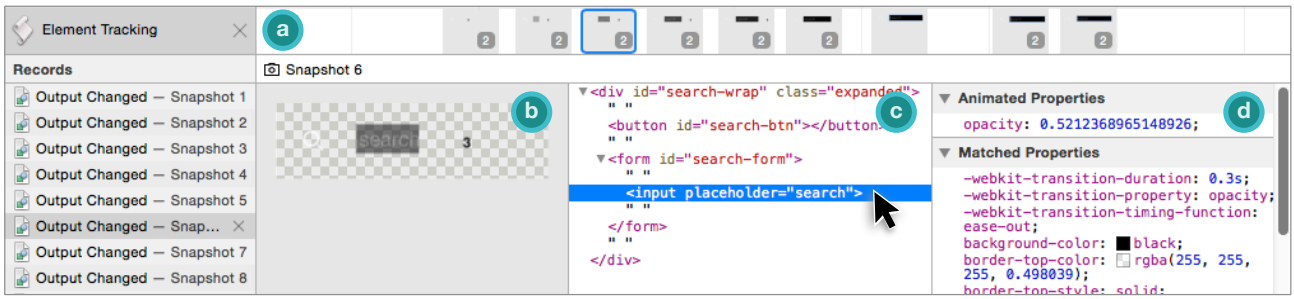

Figure 3. Scry's snapshot interface shows multiple screenshots in a timeline view (a). When the developer selects an output example from the timeline, Scry shows three views for it: (b) the element's visual appearance, (c) its corresponding DOM tree, and (d) computed style properties for the selected DOM node.

from which users can plug in their own search and navigation strategies [18] to find other relevant code upstream from the mutation operation. Initially, the user is presented with a list of recorded operations for one state difference; the user can browse these operations to understand how the state changed. Once the user wants to see the source code responsible for these mutation operations, they click on a specific operation to see where it was performed. Many mutation operations originate from source code (Figure 5), such as JavaScript function calls or assignments that cause some change to the DOM or CSS. Since these mutations may happen indirectly—for example, by adding a class, setting `node.innerHTML`, or by changing styles from JavaScript—there can be multiple JavaScript statements responsible for a change.

Capturing Changes to Visual Appearance

Scry automatically tracks changes to a user-specified DOM element's appearance and summarizes the element's output history with a series of screenshots. To start tracking an element, a developer first locates a *target element* of interest, using existing tools such as an element inspector or DOM tree viewer. Once the developer issues Scry's "Start Tracking" command (Figure 2.a), Scry immediately begins capturing a log of mutation operations for the entire document. When Scry detects changes in the target element's visual appearance, it captures a state snapshot and adds a screenshot to the target element's tracking timeline. (We later explain how these tracking capabilities are implemented.)

The element tracking timeline (Figure 3.a) is Scry's primary interface for viewing and selecting output examples. It juxtaposes these output examples—previewable screenshots of the target element—with existing timelines for familiar run-time events such as network activity, script execution, page layout, and asynchronous tasks. Timelines show events on a linear time scale and can be panned, zoomed, and filtered to focus on specific interactions or event types.

Relating Output to Internal States

Scry's snapshot interface enables a developer to learn how an element's visual appearance is rendered by juxtaposing inputs and outputs of the browser's rendering algorithm[3]. After a developer has captured relevant output states of the target element, she then selects a single screenshot from the timeline (Figure 3.a) to see more details about that visual state. The visual output, DOM subtree, and computed CSS styles for a single visual state are shown together in the snapshot detail view (Figure 3). To help a developer understand how the visual output was rendered, the visual output and CSS views are linked to the DOM tree view's current selection. When a developer selects a DOM element (Figure 3.c), Scry shows the element's matched CSS styles (Figure 3.d).

Comparing Internal States

Using Scry's comparison interface (Figure 4), a developer can quickly compare internal states of two relevant output examples to learn why the examples were rendered differently. Scry automatically discards CSS styles that are overridden in the rule cascade. Thus, the differences in two snapshots' input data—its CSS styles and DOM tree—are sufficient to explain differences in their output data.

The comparison interface (Figure 4) consists of two side-by-side snapshot interface views with additional annotations to indicate the nature of their differences. Additions are annotated with green highlights, and only appear within the temporally-later snapshot. Removals are annotated with red highlights, and only appear within the earlier snapshot. Modifications—a combination of an addition and removal for the same style property or DOM attribute—appear in purple highlights for both snapshots. Elements whose parent has changed are highlighted in yellow, and elements whose matched CSS styles have changed are rendered in bold text. As with the single snapshot view, a developer can inspect a DOM tree element to see its matching CSS styles and position within visual output. In the comparison tool, the view state of both sides is kept in sync so that the element is selected (if present) in both snapshots. This allows the developer to easily compare CSS styles and DOM states without having to recreate the same view for the other snapshot.

Relating State Differences to JavaScript Code

To complete the link from output examples to JavaScript, Scry computes which mutation operations were responsible for producing specific CSS or DOM state differences. To view the mutation operations for a difference, a developer selects a colored highlight from the comparison interface (Figure 4.a).

[3]Scry does not directly explain causal relationships between inputs and outputs in the style of Whyline [16]. Instead, Scry helps a developer, who has a working understanding of CSS-based layout, by providing concrete data against which they can validate their mental model of layout.

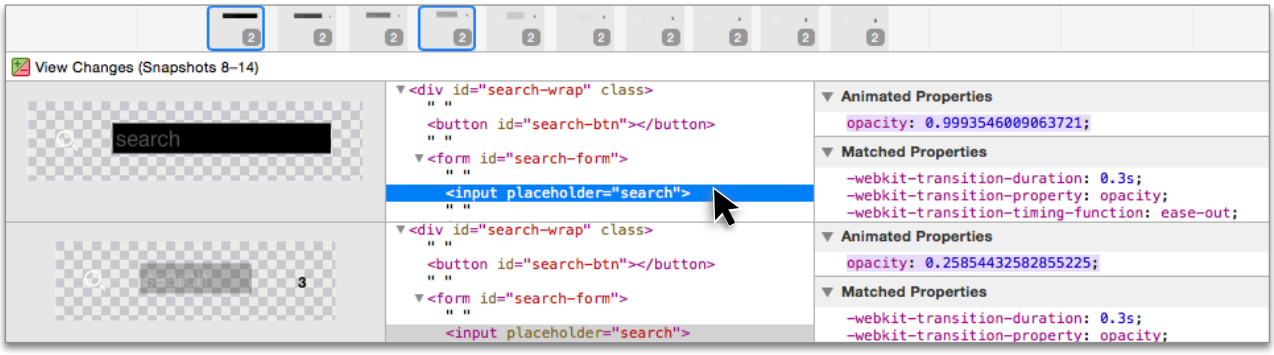

Figure 4. The snapshot comparison interface. Two screenshots are selected in the timeline, and their corresponding CSS styles and DOM trees appear below. Differences are highlighted using inline annotations; here, the `opacity` style property has changed. Additions, removals, and changes are highlighted in green, red, and purple, respectively.

Then, Scry changes views to show the difference alongside a list of mutation operations (Figure 5.b) that caused the difference. Each operation includes a JavaScript stack trace that shows the calling context for the mutation operation (Figure 5.c). Using this link, a developer can find pieces of code related to a single visually-significant difference.

IMPLEMENTATION

Scry's functionality is realized through four core features: detecting changes to an element's visual output; capturing input/output state snapshots; computing fine-grained differences between state snapshots; and capturing and relating mutation operations to state differences.

Detecting Changes to Visual Output

A central component of Scry's implementation is the *state snapshot*, which represents the state of a particular DOM element at a particular point in a program's execution. Before we discuss the data a state snapshot contains and how it is captured, we first discuss how Scry decides when to capture a snapshot of a distinct visual state.

Scry differs from prior work [24] in that it observes actual rendered visual output to detect changes to specific interface elements. When visual output significantly differs, Scry captures and commits a state snapshot. While many input state mutations may occur while JavaScript is running on the page, it is essential to Scry's example-oriented workflow that it only captures states that are visually distinct and are relevant to the target element. To detect these distinct visual states, Scry intercepts paint notifications from the browser's graphics subsystem and applies image differencing to the rendered output of the DOM element selected by the user. If the painted region does not intersect the selected element's bounding box, then Scry knows the element was not updated; if the target element's bounding box differs from its previously observed bounding box, then a snapshot is taken, as its location has moved. To check for output changes, Scry renders the target element's subtree into a separate image buffer and then compares the image data to the most recent screenshot. If the bitmaps have nontrivial differences[4] then Scry takes a full state snapshot of the target element and commits it as a distinct visual state.

Rendering and comparing an element's DOM subtree in isolation is surprisingly difficult due to two features of CSS: stacking contexts and transparency. Stacking contexts allow an element's back-to-front layer ordering to be changed by CSS properties such as `opacity`, `transform` or `z-index`. In practice, this can cause ancestor elements to be rendered visually in front of descendant elements and occlude any subtree changes. Scry mitigates this by not rendering ancestor elements and visualizing the target element's bounding box before tracking it. This strategy has shortcomings, however: descendant elements frequently allow ancestor elements to "shine through" transparent regions in order to provide a consistent background color. If ancestors are not rendered, then screenshots will lack the expected background color. To work around this, Scry retains screenshots of the target element with and without ancestor elements if they differ; the background-less version is used to detect visual changes, while the background-included version is shown to the user.

Capturing State Snapshots

When Scry decides to capture a state snapshot, it gathers many details to help a developer understand the state of the selected element in isolation from the rest of the page. Snapshots consist of the screenshot of the element's visual state in bitmap form, the subtree of the DOM rooted at the target element, and the *computed style* for each tree element. Snapshots are fully serialized in order to isolate past visual state snapshots from subsequent mutation operations.

An element's computed style describes the set of properties and values that are ultimately passed forward (but not necessarily used) in the rendering pipeline to influence visual output. In order to trace computed style property values back to specific style rules, inline styles, and mutation operations, Scry performs its own reimplementation of the CSS cascade that tracks the origin of each computed style property. Computed

[4]To compute image differences, Scry computes the mean pixel-wise intensity difference over the entire bitmap, and uses a threshold of 1% maximum difference. This allows for minor artifacts arising from subpixel text rendering and other nondeterministic rendering behavior.

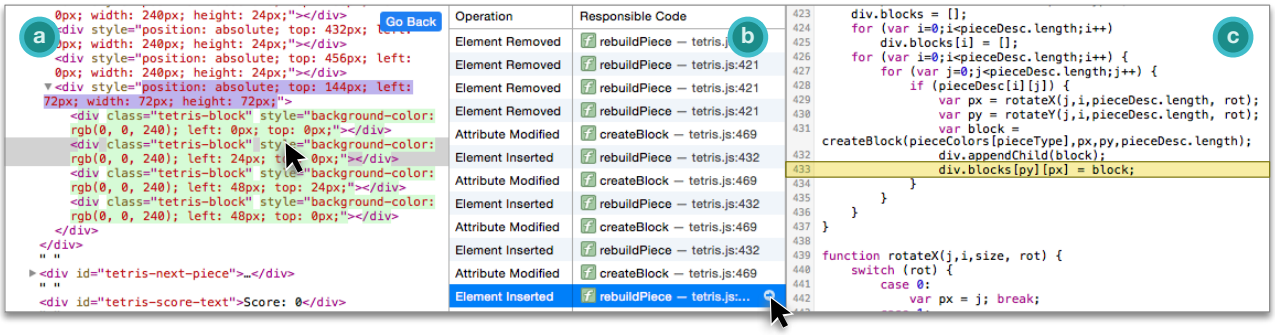

Figure 5. Explaining a DOM tree difference in a Tetris game that modifies, removes, and rebuilds DOM elements. The left pane (a) shows a single difference being investigated (the second added `<div>`). The center pane (b) shows a list of mutation operations that caused the change. When an operation is selected, Scry shows a source code preview (c) and call stack (not shown) for the operation.

style properties originate from one of four sources: declarative *style rules*, explicit *inline styles*, CSS animations, and inherited properties. In order to later deduce why a style property has changed, Scry saves the CSS rules and specific rule selectors that match each node in the snapshot.

The current Scry implementation does not attempt to capture all of a page's view state (scroll positions, keyboard focus, etc.) or external constraints (window size, locale) in state snapshots. The only exceptions are the CSS pseudo-states `:hover` and `:focus` because they are frequently used by interactive behaviors. If changes to the page's view state cause the target element's appearance or bounding box to change, then Scry will commit a new state snapshot, but it will not have sufficient information to explain how the outputs differ in terms of inputs. Prior work [6] has demonstrated that these view state inputs can be easily and cheaply collected. Insomuch as these inputs affect the set of active CSS rules, they can be treated similarly to inherited style attributes on the target element that may have global effects.

Comparing State Snapshots

Scry's usefulness as a feature location tool hinges on its ability to compute comprehensible state changes between snapshots and relate these to concrete mutation operations and JavaScript code. To precisely compare two snapshots, Scry compares each snapshot's 1) captured DOM subtrees and 2) computed styles. In the remainder of this section, we refer to the two snapshots being compared as the *pre-state* and *post-state*.

DOM Trees

Scry compares DOM subtrees and computes change summaries on a per-node basis. To compute an element's change summary, Scry first finds the element in both snapshots. To do this, Scry associates a unique, stable identifier with each DOM node at run time to make it possible to find the same node in two snapshots via a hash table lookup. If Scry finds the corresponding nodes in both snapshots it summarizes differences in their parent-sibling relationships, attributes, and computed styles. A node that appears in only one snapshot is reported as added or removed, and a node whose parent changed or whose order among siblings changed is reported as moved. This strategy identifies many small, localized changes (Table 2) that are straightforward to explain in terms of low-level mutation operations (Table 1). Moreover, these summaries correspond to the types of changes that developers are accustomed to reading in text diff interfaces, making them familiar and easy to comprehend.

Input affected	Data affected	JavaScript API / change origin
DOM	Tree Structure	`Node.appendChild`
	Node Attributes	`Node.className`
	Node Content	`Node.textContent`
	Bulk Subtree	`Node.innerHTML`
CSS	Style Rules	*various*
	Inline Styles	`Element.style`
	Animated Properties	`animation` CSS property
	Legacy Attributes	`Element.bgcolor`
View State	Scroll Positions	*user*, `Node.scrollOffset`
	Mouse Hover	*user*
	Keyboard Focus	*user*
Environment	Window Size	*operating system*

Table 1. Input mutation operations. View State and Environment are not currently supported in Scry, but are listed for completeness.

An alternative strategy for comparing subtrees is to globally summarize changes using tree matching algorithms [17] or tree edit distance algorithms [3]. We found these to be unsuitable for linking small state changes back to JavaScript code. Tree matching algorithms compute per-node similarity metrics, but do not try to attribute per-node dissimilarities to mutation operations. Edit distance algorithms do not directly produce per-node change summaries, and describe mutations using a minimal sequence of abstract tree operations. Web pages' mutation operations do not correspond to tree edit script operations: real edit sequences are often not minimal (for example, repeated mutations of a node's `class` attribute should not be coalesced) and include redundant but useful operations (such as replacing a subtree by assigning to `Node.innerHTML`).

Computed Styles

To compute differences between a single node's computed styles in the pre-state and post-state, Scry uses set operations on CSS property names. To determine which properties were added or removed, it computes the set difference. Property names present in both snapshots are compared to detect whether their property values or origins differ.

Explaining State Differences

Input affected	Change type	Cases
DOM	Node Existence	node-added, node-removed
	Relationships	parent-changed, ordinal-changed
	Attributes	attribute-changed, attribute-added, attribute-removed
CSS	Property Existence	property-added, property-removed
	Direct Styles	value-changed
	Indirect Styles	origin-changed

Table 2. Possible cases for per-node change summaries produced by comparing state snapshots. Similar cases are shown the same way in the user interface, but are summarized separately to simplify the task of finding a corresponding direct mutation operation.

When a user selects a specific state difference to see what code was responsible, Scry presents a sequence of JavaScript-initiated mutation operations that caused the difference. Scry computes this causal chain on-demand in three steps. First, using the affected node's change summary, Scry finds a single *direct operation* within its operation log that produces the node's expected post-state. Second, Scry finds multiple *prerequisite operations* which the direct operation depends on. Lastly, the operations are ordered and presented in the user interface as a causal chain connecting the node's pre-state and post-state.

As a starting point, we first discuss the mutation operations that Scry captures as raw material for producing causal chains. Then, we detail the specific strategies that Scry uses to identify the code responsible for a change: (1) how to identify direct operations for node changes and simple style changes; (2) how to find direct operations that indirectly cause computed styles to change via style rules; (3) and how to compute dependencies between mutation operations.

Capturing Mutation Operations

The web exposes a large, overlapping set of APIs to effect changes to visual appearance by mutating rendering inputs. This section enumerates these input *mutation operations* that Scry must log and relate to state differences. Scry instruments APIs and code paths for each of the input mutation operations listed in Table 1. While tracking a target element, Scry saves a log of these mutation operations for later analysis. At the time that each mutation operation is logged, if the operation is performed by JavaScript code, Scry also captures a call stack, to help the developer link state differences to JavaScript code causing the mutation, and the upstream code and event handlers that caused it to execute.

Mutation operations as defined by Scry (Table 1) closely mirror the most commonly used DOM and CSS APIs. These operations can be used to explain changes to DOM state, and changese to computed style properties that originate from style rules (whose rules match and unmatch as the DOM tree changes). Scry also captures mutation operations from other computed style property change origins, such as an element's animations and inline styles set from JavaScript codoe.

Finding Direct Operations for DOM Changes

For a specific state change (Table 2), Scry scans backwards through the operation log to find the most recent operation related to the state change. The most recent operation that mutates state into the post-state is the change's direct operation; other prerequisite operations are separately collected as the direct operation's dependencies (described below). For attribute differences, Scry finds the most recent change to the attribute. For tree structure differences, Scry determines what operations could have caused the change and finds the most recent one with the correct operands. For example, if a node's ordinal rank among its siblings differs, then Scry looks for operations that inserted or removed nodes from its parent.

Finding Direct Operations for Style Changes

Scry uses origin-specific strategies to find direct operations for a computed style property change. If a property originates from an inline style that was set from JavaScript, Scry simply scans backwards for a mutation operation that directly assigned that inline style. If a property originates from a declarative CSS animation or transition, then the browser rendering engine automatically changed the property value, triggered by an element gaining or losing a `animation` property from its computed style. In this case, the user wants to know where the originating `animation` property came from, so Scry finds the direct operation that caused the animation property to change.

If a property originates from a style rule, then Scry must determine which of the element's matched rules changed and relate that to a DOM difference. Properties can be added or removed when rules start or stop matching the element. Changes to a property's value may happen when rules either match or unmatch and change the results of the CSS cascade. Therefore, Scry analyzes how a node's matched rules and selectors differ between snapshots to find what caused different rules to match. To change result of the CSS cascade, either a rule must stop matching and "lose" the property, or a rule must start matching and "win" the property. If the losing rule is not present in the post-state, then Scry looks for state differences between the snapshots that could cause the selector to no longer match. For example, if the rule `div.hidden { display: none; }` stopped matching a `<div>` element, then Scry deduces that a differing `class` attribute caused the rule to stop matching.

Computing Dependencies between Mutation Operations

To provide the user with a sequence of operations that transform the pre-state into the post-state, Scry must compute dependencies between mutation operations. This is similar to the notion of an executable *program slice*: the operation sequence must preserve a specific behavior (cf. a *slicing criterion*), but it is permissible for it to over-approximate and include irrelevant operations. Reducing the operation trace length (cf. slice size) for a state change simply makes it easier for a human to browse and comprehend how the change occurred. Note that these dependencies only ensure that the mutation operations preserve the specific state changes captured in the pre-state and post-state[5].

[5] Since JavaScript can access DOM state and layout results, there are untracked control and data dependencies between JavaScript and inputs. We leave dynamic slicing of JavaScript dependencies to future work.

Scry computes an operation dependency graph on-demand as a user selects pre-state and post-state snapshots. To produce a causal chain for a change, Scry finds the change's direct operation in the dependency graph, collects operations its transitive closure, and orders operations temporally. Dependencies for operations between the pre-state and post-state are computed in three steps: first, operations are indexed by their node operands. Second, Scry builds a directed acyclic graph with operations as nodes and causal dependencies between operations as directed edges. Operations that do not explicitly depend on other operations implicitly depend on the pre-state. Scry processes operations backwards starting from the post-state; each operation's dependencies are resolved in a depth-first fashion before processing the next most-recent operation. Finally, when all operations have been processed, graph nodes with no outgoing edges (i.e., depend on no other operations) are connected to a node representing the pre-state.

Operations that mutate node attributes and inline styles require the operand nodes and attributes to exist. For example, an attribute-removed operation depends on the existence of a node n and attribute a to remove. If neither n or a existed in the pre-state, then the operation's dependencies include the subsequent mutation operations that created n and/or a. Similarly, operations that change the structure of the DOM (append-child, set-parent, replace-subtree, etc.) require all of their operands to exist.

Prototype Implementation

Scry is implemented as a set of modifications to the WebKit browser engine [30] and its Web Inspector developer tool suite. To provide the element tracking user interface, Scry extends the Web Inspector with a new screenshot timeline, snapshot detail and comparison views, and integrations between difference summaries and other parts of the interface. Scry also tracks dependencies for mutation operations and finds direct mutation operations in the JavaScript-based frontend. Element screenshots, DOM tree snapshots, style snapshots, and mutation operations are gathered through direct instrumentation of WebKit's WebCore rendering engine and sent to the Web Inspector frontend. Scry tabulates computed styles in C++ with full access to the rendering engine's internal state.

PRACTICAL EXPERIENCE WITH SCRY

Despite the rise of a few dominant client-side JavaScript programming frameworks, web developers use DOM and CSS in endlessly inventive ways that tool developers cannot fully predict. In our experience, even when Scry's results are diluted by idiosyncratic uses of web features, it is still helpful for at least *some* parts of a feature location task. This section presents several short case studies that illustrate Scry's strengths and weaknesses, motivating future work.

Expanding Search Bar

A National Geographic web article[6] contains a navigation bar with an expanding search field. When the user clicks on the magnifying glass icon, a text field appears and grows to a reasonable size for entering search terms. Without Scry, this behavior is difficult to investigate because the animation lasts less than a second, and intermediate animation states are not displayed or persisted.

To understand this widget, we used the Web Inspector's "Inspect" chooser to locate the search icon element in the DOM tree browser. We then started tracking the element with Scry and interacted with the widget to start its animation. Upon browsing captured screenshots, we saw that the text field's width and opacity both changed. We compared two snapshots with Scry and saw that separate CSS `transition` properties were applied to different tree elements. We clicked on the animated property value and Scry presented a list of mutation operations, revealing that a `click` event handler had added a `.expanded` class to the root element of the widget to trigger an animated transition. In this example, Scry was particularly helpful in two cases: (1) it captured intermediate animated property values which are normally not possible to see in the inspector; and (2) Scry was able to trace the cause of the entire animation back to a single line of JavaScript code that changed an element's class name.

A Tetris Clone

A Tetris-like game[7] uses DOM elements and CSS to render the game's board and interface elements. To understand how the board is implemented using CSS, we used Scry to track changes to the main playing area. As we played the game, Scry took full snapshots of the game board. By inspecting the DOM of each snapshot, it became apparent that the game board is implemented with one container element per row and multiple square-shaped `<div>`s per row to form pieces. When we compared two board states that had no pieces in common, we unexpectedly found that Scry identified two squares on the board as being the same. After following mutation operations into the JavaScript implementation, we discovered that the Tetris game uses an "object pooling" strategy. To produce shapes on the game board using squares, the game reuses a fixed set of DOM elements and explicitly positions them using inline styles. Scry's confusion arose because the game board states happened to reuse the same square elements from the object pool.

From this example, we learned that although Scry's current implementation expects that each allocated DOM element has a consistent identity, many applications violate this expectation. Some client-side rendering frameworks such as React [13] expose an immediate-mode API called the *virtual DOM*. Client JavaScript implements `draw()` methods that fully recreate a widget's DOM tree and CSS styles using the virtual DOM. Behind the scenes, React synchronizes the virtual and real DOM using a fast tree edit algorithm, reusing the same elements to produce visual output for unrelated model objects that happen to use the same HTML tag names. A similar problem arises with frameworks that re-render a component by filling in an HTML string template and overwriting the component's prior DOM states by setting the target element's `innerHTML` property. In this case, Scry shows the target element's entire subtree as being fully removed and re-added. Scry doesn't try to match

[6] National Geographic, *Forest Giant*. **http://webplatform.adobe.com/Demo-for-National-Geographic-Forest-Giant/browser/src/**

[7] Tetris Clone. **http://timothy.hatcher.name/tetris/**

similar nodes, but could be extended to fall back to using more relaxed similarity-based metrics [17] instead of strict identity when re-finding DOM elements.

A Fancy Parallax Demo

In the past few years, browsers added support for applying 3D perspective transforms to elements using CSS. The fancy parallax demo[8] discussed here is representative of pages that use scroll events and transforms to implement parallax and infinite scrolling effects. For this page, we wanted to learn how an element's position is computed in response to scrolling events. We used Scry to track an animated paragraph of text as it moved around when we scrolled the page. From a single DOM tree snapshot, we could see that CSS `transform`-related properties were set on all elements subject to scroll-driven animations. We compared two snapshots to find the source of changes to the `transform` properties, and were always led back to the same line of JavaScript code. Looking upstream in the stack trace, it appeared that a JavaScript library interprets the single scroll position change and imperatively updates the `transform` style property for dozens of elements. We could not discover (using Scry) where the animation configurations for each element were specified.

From this example, we learned that Scry is of limited use for localizing code when the endpoints of JavaScript—where it directly interfaces with rendering inputs—are not easily distinguishable. Such *megamorphic* callsites to browser APIs are common when a web page calls DOM APIs indirectly through utility libraries. A simple solution would be to automatically disambiguate very active callsites based on their calling context, or allow the user to hide library code (known as "script blackboxing" in some browsers). However, for this example, simply filtering the stack traces would not lead a user to the configuration data for a parallax animation. A better solution would be to extend Scry's capabilities to include tracking of control and data dependencies through JavaScript [26]. This would require a very different technical approach, since Scry instruments native browser APIs rather than JavaScript code.

RELATED WORK

Scry aids a developer in locating the code that implements a feature, and in understanding how an interactive behavior is implemented in terms of CSS, DOM, and JavaScript. In the software engineering research literature, these two activities—known as *feature location* [11] and *program comprehension* [10]—are the subject of entire subfields and have been explored using a wide variety of techniques. Below, we discuss Scry's influences and compare its functionality to similar tools.

Locating Features using Visual Output

Scry is uncommon among feature location tools in that uses pixel-level visual states as input specifications for a feature. Tools for selecting features based on their output are particularly useful for user interfaces or graphically intensive software such as video games, web sites [7], and visualizations. This is because many features (and bugs) have obvious visual manifestations which are easier to find than a feature's small, subtle internal states. Visually-oriented runtime environments such as web browsers and the Self VM [29], have long supported the ability to introspect interface elements from the program's current visual output and vice-versa. Scry extends this capability to also support inspecting and comparing snapshots of past interface states.

[8]Fancy Parallax Demo. `http://davegamache.com/parallax/`

Explaining How Interactive Behaviors Work

Scry follows a long line of methods and tools [27] that aim to help a developer comprehend specific program features or behaviors. Recent work for user interfaces has focused on inferring behavioral models [19, 1, 21], logging and visualizing user inputs and runtime events [24, 6], and using program analysis to produce causal explanations of behaviors [16, 26]. Scry shares the same reverse-engineering goals as FireCrystal [24], which also logs and visualizes DOM mutations that occur in response to user interactions. However, FireCrystal reveals all mutation operations on a timeline without any filtering, which quickly overwhelms the user with low-level details. Scry supports a staged approach to comprehension by presenting self-contained input/output snapshots and only showing the mutation operations necessary to explain a single difference between snapshots.

Scry's example-oriented explanations are similar to those produced by Whyline [16]. Whyline suggests context-relevant comprehension questions that it is able to answer, whereas Scry enhances a user's existing information-seeking strategies by providing otherwise-inaccessible information. Whyline and other tools based on dynamic slicing [31] may provide more comprehensive and precise explanations than Scry, but require expensive runtime instrumentation that limits the situations in which these tools can be used. In a different approach to making short explanations, recent work on observation-based slicing [4, 32] proposes to minimize inputs to a rendering algorithm while preserving a subset of the resulting visual output. Scry could use this approach to reduce snapshots by discarding apparently "ineffective" style properties that have no visual effect.

DISCUSSION AND FUTURE WORK

Scry is a first step towards demystifying the complex, hidden interactions between the DOM, CSS layout, JavaScript code, and visual output. The capabilities we have described in this paper validate our interface concept; other explanatory capabilities could be added without significantly altering Scry's staged, example-oriented workflow. In particular, we see two promising directions for future work: expanding the scope and accuracy of Scry's explanations, and tracking an element's changes backward (rather than only forward) in time.

Opening the Layout Black Box Scry treats the layout/rendering pipeline as a black box, but users often want to know how single style properties are used (or not) within the pipeline. Within the design space of black-box approaches, it would be straightforward to extend Scry to further minimize rendering inputs using observation-based slicing [4]. Concretely, Scry could delete individual style properties from a snapshot if the resulting visual output does not differ [32]. Even with a minimal set of inputs, a more involved "white-box" approach to

explaining layout [22] would still be extremely useful. By adding more instrumentation to browser rendering engines, Scry could be extended to directly answer why and why-not questions [15, 22, 16] about how inputs are used or how outputs are derived as they funnel through the increasingly complex layout algorithms of modern web browsers.

Tracking Causality through JavaScript Scry can explain DOM or CSS differences in terms of mutation operations, but it does not track the upstream dependencies in JavaScript code that caused the mutation operations. Scry could be extended with recent work on JavaScript slicing [26] and event modeling [1] to extend its explanations to show an uninterrupted causal chain [16] between user inputs and events, JavaScript state and control flow, mutation operations, and changes to layout inputs and outputs. This would produce explanations of changes that would be both more complete and more precise.

Tracking Past Element States Given a target element, Scry can track its future visual states as a developer demonstrates the behavior of interest by interacting with the page. However, in fault localization tasks a developer often wants to see what went wrong in the past that produced a buggy state in the present. To gather past states of an element, Scry could build on recent deterministic replay frameworks for web applications [6] to collect snapshots and trace data from earlier instants of the execution. Prior work has demonstrated the feasibility of such an "offline dynamic analysis" [8, 26, 9], but none has integrated this technique into a user interface or web browser.

REFERENCES

1. Alimadadi, S., Sequeira, S., Mesbah, A., and Pattabiraman, K. Understanding JavaScript event-based interactions. In *ICSE* (2014).
2. Andrica, S., and Candea, G. WaRR: A tool for high-fidelity web application record and replay. In *DSN* (2011).
3. Bille, P. A survey on tree edit distance and related problems. *Theoretical Comput. Sci. 337* (June 2005).
4. Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., and Yoo, S. ORBS: language-independent program slicing. In *FSE* (2014).
5. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI* (2009).
6. Burg, B., Bailey, R. J., Ko, A. J., and Ernst, M. D. Interactive record and replay for web application debugging. In *UIST* (2013).
7. Chilana, P. K., Ko, A. J., and Wobbrock, J. O. LemonAid: selection-based crowdsourced contextual help for web applications. In *CHI* (2012).
8. Chow, J., Garfinkel, T., and Chen, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX* (2008).
9. Chow, J., Lucchetti, D., Garfinkel, T., Lefebvre, G., Gardner, R., Mason, J., Small, S., and Chen, P. M. Multi-stage replay with Crosscut. In *VEE* (2010).
10. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. A systematic survey of program comprehension through dynamic analysis. *IEEE TSE 36* (Sep./Oct. 2009).
11. Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process 25* (2013), 53–95.
12. Eckert, C., and Stacey, M. Sources of inspiration: a language of design. *Design Studies 21*, 5 (2000), 523–538.
13. Facebook. React: a JavaScript library for building user interfaces, 2015. `https://facebook.github.io/react/index.html`.
14. Hilgart, M. Step-through debugging of GLSL shaders, 2006.
15. Ko, A. J., and Myers, B. A. Designing the WhyLine: a debugging interface for asking questions about program behavior. In *CHI* (2004).
16. Ko, A. J., and Myers, B. A. Extracting and answering why and why not questions about Java program output. *ACM TOSEM 20*, 2 (Sep. 2010), 1–36.
17. Kumar, R., Talton, J. O., Ahmad, S., and Klemmer, S. S. Bricolage: example-based retargeting for web design. In *CHI* (2011).
18. Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S. D. How programmers debug, revisited: An information foraging theory perspective. *IEEE TSE 39*, 2 (Feb. 2013), 197–215.
19. Mesbah, A., van Deursen, A., and Lenselink, S. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM TWEB 6*, 1 (2012), 1–30.
20. Mickens, J., Elson, J., and Howell, J. Mugshot: deterministic capture and replay for JavaScript applications. In *NSDI* (2010).
21. Mirzaaghaei, M., and Mesbah, A. DOM-based test adequacy criteria for web applications. In *ISSTA* (2014).
22. Myers, B., Weitzman, D. A., Ko, A. J., and Chau, D. H. Answering why and why not questions is user interfaces. In *CHI* (2006).
23. O'Callahan, R. Efficient collection and storage of indexed program traces, 2006. `http://www.ocallahan.org/Amber.pdf`.
24. Oney, S., and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *VL/HCC* (2009).
25. Röthlisberger, D. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, Universität Bern, 2010.
26. Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE* (2013).
27. Storey, M.-A. Theories, methods, and tools in program comprehension: Past, present, and future. In *IWPC* (2005).

28. Stylos, J., and Myers, B. A. Mica: a web-search tool for finding API components and examples. In *VL/HCC* (2006).
29. Ungar, D., Lieberman, H., and Fry, C. Debugging and the experience of immediacy. *CACM 40* (Apr. 1997), 38–43.
30. WebKit contributors. The WebKit open source project, 2012. `http://www.webkit.org/`.
31. Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. A brief survey of program slicing. *ACM Softw. Eng. Notes 30*, 2 (2005).
32. Yoo, S., Binkley, D., and Eastman, R. Seeing is slicing: Observation based slicing of picture description languages. In *SCAM* (2014).