Understanding Dynamic Behavior with Tools for Retroactive Investigation

Brian Burg

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Supervisory Committee:

Michael D. Ernst, Chair

Andrew J. Ko

Steve Tanimoto

James Fogarty

Sean Munson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Understanding Dynamic Behavior with
Tools for Retroactive Investigation

Brian Burg

Chair of the Supervisory Committee:
Associate Professor Michael D. Ernst
Computer Science and Engineering

The web is a widely-available open application platform, where anyone can freely inspect a live program's client-side source code and runtime state. Despite these platform advantages, understanding and debugging dynamic behavior in web programs is still very challenging. Several barriers stand in the way of understanding dynamic behaviors: reproducing complex interactions is often impossible; finding and comparing a behavior's runtime states is time-consuming; and the code that implements a behavior is scattered across multiple DOM, CSS, and JavaScript files.

This dissertation demonstrates that these barriers can be addressed by new program understanding tools that rely on the ability to capture a program execution and revisit past program states within it. We show that when integrated as part of a browser engine, deterministic replay is fast, transparent, and pervasive; and these properties make it a suitable platform for such program understanding tools. This claim is substantiated by several novel interfaces for understanding dynamic behaviors. These prototypes exemplify three strategies for navigating through captured program executions: (1) by visualizing and seeking to inputs—such as user interactions, network callbacks, and asychronous tasks; (2) by retroactively logging program states and seeking execution back to log-producing statements; and (3) by working backwards from differences in visual output to the source code responsible for inducing output-affecting state changes. Some of these capabilities have been incorporated into the WebKit browser engine, demonstrating their practicality.

ACKNOWLEDGMENTS

The author thanks everyone for being awesome. todo ►*rewrite.*◄

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The world wide web's rise as *the* platform-independent application runtime has democratized the development of documents, applications, and user interfaces. Unlike on most platforms, web programs are transmitted in source form as HTML, CSS, and JavaScript code. Using web developer tools included with most web browsers, a user can inspect and modify a client-side web program's source code and runtime states. This capability dramatically lowers technical barriers to finding interesting behaviors within third-party web content and inspecting the corresponding source code for a behavior.

Code that implements designs and behaviors is readily available on the web, but is just as difficult to understand and debug as code written for any other platform.[1] Modern web programs are complex, interactive applications built using a combination of several tools, frameworks, languages, and other technologies. While originally intended for static documents, web technologies such as HTML, DOM, JavaScript and CSS are now the building blocks for large cross-platform application suites developed by teams of software engineers. Beyond the challenges inherent to building user interfaces, the interactions between web technologies introduce significant incidental complexity. The combination of declarative CSS styles, imperative JavaScript code, and a retained document rendering model can obfuscate a program's dependencies and causal relationships [**?**].

Existing developer tools are inadequate for debugging a web program's interactive behaviors. Many tools distributed with web browsers simply reimplement "baseline" tools from other development environments, such as logging, profiling, and breakpoints. These tools are ill-suited for understanding nondeterministic, interactive, and time-sensitive behaviors that are pervasive in

[1] Software maintenance tasks—a category which includes debugging—is commonly observed to comprise 50–80% of a developer's time. [**?**]

client-side web programs. The complexity of web programs is often limited by developers' ability to debug and maintain these systems [**?**] with poorly-suited tools. Using application frameworks and code conventions can mitigate the incidental complexity of writing these programs; however, well-disciplined code can sometimes be even more difficult to understand at first glance [**?**].

1.1 The Problem

Several barriers stand in the way of understanding dynamic behaviors: reproducing complex interactions is often impossible; obtaining runtime states relevant to a behavior is time-consuming; and the code that implements a behavior is scattered across multiple DOM, CSS, and JavaScript files. Underlying these issues is a common problem: *existing tools allow a user to inspect current or future program states, but not past program states*. Thus, a user must configure these tools—such as breakpoints or logging—before the behaviors of interest actually occur. This places a considerable burden on the user, who must correctly predict what runtime states might be useful (and how to gather them) in order for the tools to have any benefit. In the face of nondeterministic behavior, repeatedly reproducing the same behavior after changing tool configurations is error-prone or impossible. Even when behavior is deterministic, logging desired runtime states or pausing the debugger at an important statement requires careful planning and time-consuming experimentation. Debuggers, profilers, and other tools that support inspection of live runtime information during execution are often inappropriate for investigating interactive behaviors, because these tools may slow or suspend execution and thus spoil the behavior or the runtime data. Lastly, instead of navigating backwards from current program states to previously-executed code responsible for the state—working from effects back to causes—a user must configure their tools to gather information at an earlier instant. Effectively, a user must predict cause-effect relationships responsible for program states and confirm their guesses by observing actual runtime behaviors.

1.2 Addressing the Problem

Web developers are hamstrung because existing tools are necessarily limited to inspecting future program states. What if a developer could gather, visualize and revisit *past* program states after they

already occurred, rather than configuring tools for current and future program states? What if it were possible to capture a single execution and retrieve runtime states from it as needed, rather than guessing ahead of time what program states might be useful to log? Consider the task of debugging an interaction in a video game: instead of manually reproducing game behavior whenever different runtime information is desired, a developer could play the game once and later go "back in time" in the captured execution to examine past runtime states. This workflow avoids repetitious, error-prone gameplay and decouples playing the game from interruptions such as setting up logging, turning breakpoints on and off, or performance slowdowns from heavyweight instrumentation [**?**].

In this dissertation, I investigate how this *retroactive* approach to program understanding can be realized through novel runtime techniques, user interfaces, and integrations with new and existing developer tools. In particular, I claim the following thesis statement:

> *The ability to revisit past program states enables new tools for understanding dynamic behaviors in web programs, and browser engines can provide this capability through fast, transparent, and pervasive deterministic replay.*

I substantiate this claim by investigating two related lines of research: how browser engines can capture and replay web program executions using deterministic replay techniques; and what tools, strategies and interfaces are necessary to find relevant program states and facts within a captured execution. In order to revisit arbitrary past program states during program understanding tasks, it must be possible to exactly reproduce an execution on demand. The first part of this dissertation investigates how deterministic replay techniques can provide this capability for high-level managed runtimes such as browser engines. In order to work with executions as first-class objects, it must be possible for a replay-enabled developer tool to reference specific execution instants and efficiently collect historical runtime data. The middle part of this dissertation develops several such gadgets that serve as a crucial linkage between the needs of a replay-enabled tool and the capabilities of a deterministic replay infrastructure. In order to act upon past program states, it must be possible to quickly find relevant facts among the vast amount of runtime information produced during an

execution. The last part of this dissertation develops several task-oriented navigation strategies and supporting replay-enabled tools for finding relevant program states in a captured execution.

1.3 Definitions

This dissertation builds upon work from various disciplines and fields such as Human-Computer Interaction, Program Analysis, Compilers, and Software Engineering. Thus, it is useful to define and consistently use key terms that are otherwise prone to misinterpretation.

Many terms exist to categorize people who perform *programming*: instructing a computer (via code or other directives) to perform actions at a later time. This dissertation refers to any such person with the generic term *developer*. A *novice developer* has a little practice; a *skilled developer* has more than a little practice[2]; a *professional developer* is paid for his or her programming activities. An *end-user programmer* writes code only with the purpose of supporting a larger task or goal. A *tool developer* creates tools for use by *tool users*, who create programs for *end-users*.

This dissertation is primarily concerned with the family of tasks referred to as *program understanding*: any process undertaken (typically by a developer) to develop an explanation of how a program did execute, will execute, or will not execute. Specific program understanding tasks include *feature location*: developing an explanation of what code implements a specific behavior; and *debugging*: developing an explanation of undesirable or unexpected behaviors. Various debugging and bug-related terms also deserve definition. A *failure* is an undesirable program output that does not conform to the program's expected behavior. A *fault* is a program state that may lead to a *failure*. This dissertation uses the terms *error* and *defect* interchangeably to refer to parts of program code that cause a *fault* and/or *failure*. A *bug* loosely refers to single or a combination of failure(s), fault(s), and/or defect(s).

The long history and evolution of the world wide web has led to many confusing, similar terms. *A web developer* is a developer who produces web content. This dissertation uses the terms *web content*, *web program*, and *web application* interchangeably to refer to documents consisting of

[2]Where possible, scenario-relevant modifiers such as *successful* and *unsuccessful* are preferred in place of subjective or demographic-based terms such as *senior*, *novice*, and *skilled*.

HTML, CSS, DOM, JavaScript, and related technologies that are viewable by a *web browser*. A *web browser*—sometimes referred to as a *user agent* in web standards—is an end-user application for viewing web content. A *browser engine* is a managed language runtime capable of downloading, parsing, interpreting, and rendering untrusted web content, and is separate from other web browser functionality such as bookmarks, tabs, and other user interface elements. Finally, *web developer tools* are program understanding tools used by web developers. This dissertation mainly discusses web developer tools that are distributed as part of a web browser.

1.4 Contributions

This dissertation introduces several major contributions that extend the state of the art in runtime techniques and program understanding tools:

- Techniques for fast, pervasive and transparent deterministic replay of web content.
- Algorithms for revisiting any executed statement within a captured execution.
- Two visualizations of a captured execution that support navigating via top-level actions.
- An interface for retroactively logging runtime states and revisiting their execution context.
- The first user study examining the benefits, drawbacks, and design concerns for interactive record/replay user interfaces.
- Algorithms for efficiently detecting, serializing, and comparing visual states over time.
- Algorithms for establishing causality between visual changes, state changes, and code.
- An interface for feature location based on comparing output and state changes.

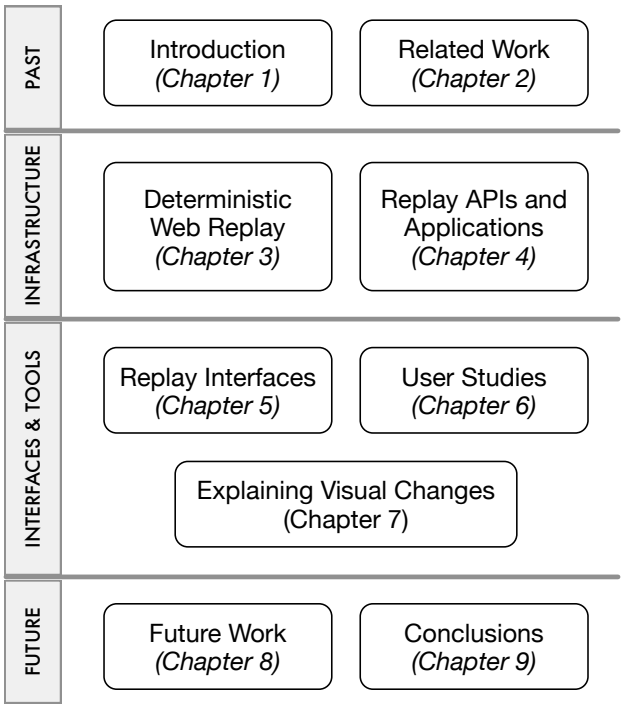

Figure 1.1: Major sections and chapters of this dissertation, arranged from beginning to end.

1.5 Outline

The remainder of this dissertation is organized as shown in Figure 1.1. Following this section is Chapter 2, which surveys prior work with a focus on techniques, tools, interfaces, and information needs that inform the above contributions. The second part describes the Dolos replay infrastructure (Chapter 3) and extensions to Dolos that support common replay use cases (Chapter 4). The third section describes a series of program understanding tools that support different navigation strategies: navigation via inputs (Chapter 5), navigation via logged outputs (Chapter 5), and navigation via visual state changes (Chapter 7). The final part of this document sketches several directions for future research (Chapter 8), and presents the conclusions of this research (Chapter 9).

Chapter 2

RELATED WORK

2.1 Reproducing and Revisiting Dynamic Behavior

2.1.1 Deterministic Replay

Deterministic replay is a well-studied technique [23, 31] for reproducing dynamic behaviors, but only recently have researchers investigated replaying interactive, user-driven programs in ways that produce full visual output. Contemporary deterministic replay research focuses on cheaply replaying multithreaded and multicore programs; virtual machine [33] and hardware support for capture and replay; adding replay support to new languages and runtimes; and novel applications of replay techniques to other domains [17, 87]. Few of these systems are production quality, and even the most robust and widely-deployed replay tools [84, 131] are designed for expert use via debugger commands (discussed in Section 2.1.3).

Prior research into deterministic replay for web applications has mostly focused on standalone tools for cross-browser replay. Mugshot [80] uses a reverse network proxy to instrument incoming web applications at the source level. It injects native function wrappers and DOM event listeners into the web application to transparently capture and replay JavaScript from "user-space". JS-Bench [103] uses the same instrumentation approach to synthesize JavaScript benchmarks from a captured execution. FireCrystal [91] is an extension for Firefox [83] that captures DOM events and attempts to replay them on an isolated copy of the web application. The WaRR [3] infrastructure captures DOM events from within WebKit and replays them through a Selenium [124] plugin.

Dolos differs from these tools by choosing integration with standard developer tools [90, 135] and high-fidelity replay as top-level design goals. Native tool integration is directly at odds with cross-browser approaches because of the sandboxed nature of web applications. A sandboxed JavaScript program (using GDB terminology [38], an *inferior process*) cannot interact with priv-

ileged JavaScript debuggers (a *superior process*), so a replay system implemented in the inferior process is unusable while the program is paused at a breakpoint. High-fidelity replay of JavaScript is not possible with "user-space" libraries and source instrumentation because not all APIs can be mediated, and many sources of nondeterminism—such as timers, animations, and resource loading—cannot be controlled from outside of the rendering engine.

While Dolos goes to great lengths to make JavaScript execution completely deterministic, this may limit a user's wish to reuse nondeterministic inputs on a different program version [17, 52, 87**?**] or intentionally diverge a replayed execution to explore alternate histories. Probe points (Chapter 5) address logging—the most common reason to alter the program—but are not designed for side-effecting operations. Scribe [60] is a multicore deterministic replay system designed to "go live" from the end of a captured execution; Dora [129] extends Scribe to meaningfully exclude portions of a recording that are affected by divergence. Another line of work attempts to deterministically reproduce outputs from anonymized logs [22], partial runtime state [50], or minimal replay logs [2].

2.1.2 Tracing and Post-mortem Techniques

A well-studied alternative to deterministic replay systems are post-mortem trace-based tools. These tools gather exhaustive traces of execution at runtime and provide affordances for querying, analyzing, or visualizing the traced behavior after the program has finished executing. We discuss several trace-based systems that are designed for program understanding tasks.

Trace-based techniques are generally avoided for JavaScript programs because the output of web applications is highly visual in nature and capturing a trace in memory can quickly make an application become unusably slow. Two exceptions are JSMeter [100] and DynJS [102, 104], which both instrument the browser itself to collect a detailed trace of JavaScript execution for offline simulation.

Amber [88] and Nirvana [7] are two tools for efficiently collecting, storing, and indexing traces from executions of x86 binaries. Both tools use dynamic binary rewriting frameworks like Valgrind [86] to instrument and capture a detailed log of register operations, and memory operations,

and control flow. Nirvana employs sophisticated compression mechanisms and partial coalescing to achieve low overhead while the program executes, but must re-simulate execution to produce accurate results. Amber incurs high capturing overhead but precomputes indexes of memory effects and requires no re-execution. Tralfamadore [69, 70] captures a low-level unindexed trace at runtime; offline, it runs a dynamic analysis over the trace in a streaming pipeline that successively transforms the trace into higher-level events that are meaningful to a user.

ODB [73] was the first trace-based omniscient debugger for Java programs. It heavily instruments JVM bytecode to record a trace of all memory activity and control flow. It uses information in the trace to populate IDE views containing value histories, rematerialized local variables, and a call stack. The omniscient debugger TOD [96, 98] improves on ODB by incorporating modern database indexing and query optimization techniques and partial deterministic replay. STIQ [97] further improves performance with optimizations for random memory inspection, causality links between reads and writes, and bidirectional debugger commands (Section 2.1.3). In order to recreate visual output for post-mortem debugging, Whyline [55] uses domain-specific instrumentation of GUI toolkit APIs in order to save a trace of relevant toolkit invocations.

2.1.3 Navigating via Debugger Commands

Text-based commands for controlling capture and replay are a common affordance [84, 130, 131], and often supplement visual interfaces for replay [73]. In 1990, Tolmach and Appel [126] first described the `reverse-step` and `reverse-continue` commands in a debugger for the ML language. The seminal work for imperative debuggers is Boothe's description of efficient counter-based bi-directional debugging algorithms [11], which includes commands for `reverse-step-into`, `reverse-step-out`, `reverse-continue`, and `reverse-watchpoint`. Arya et al. have recently proposed [5] algorithmic improvements to `reverse-watchpoint` based on decomposing large debugger commands [130] like `continue` into a sequence of `step-over` and `step-into` commands. Dolos does not implement any of these commands, but there are no known complications to doing so.

2.1.4 Navigating via Program Outputs

Timelapse's use of timelines and seekable outputs is specifically designed for casual use during program understanding tasks. It draws on a long tradition [45] of graphical history visualizations such as those used extensively in the Chronicle [41] tool. Few deterministic replay tools can seek execution directly to specific logged outputs without auxilliary use of breakpoints. The YinYang [77] live programming system is one exception; however, it depends on a restricted programming model where all operations are undoable and commutative, and does support interactive programs. Web browsers and other high-level application platforms are able to relate rendered UI elements to their corresponding source implementation or runtime objects, but this is typically limited to the currently visible output of the program.

DejaVu [52] combines affordances for replaying, inspecting, and visualizing a computer vision kernel program over time. It decouples video inputs from outputs so new versions of the kernel can be tested against the same inputs. DejaVu assumes a deterministic, functional kernel so that "checkpointing" and replaying the program is a matter of re-executing the kernel from a specific frame of the input stream.

Post-mortem, trace-based investigation tools such as WhyLine [55], TOD [98], ODB [73] support navigating through traces by selecting simulated visual output, console output, or historical values of objects. Rather than using selections as a target for re-execution, these tools search for the selected instant over a large trace, and display it in the context of related information such as a matching call stack or local variables.

2.2 Analyzing and Understanding Dynamic Behavior

Scry and Overscan draw on a vast literature of techniques for instrumenting, analyzing, and visualizing runtime data for program comprehension [24]. We summarize the most relevant work below to clarify our chosen points in the design space, and describe alternatives and their tradeoffs.

2.2.1 Implementing and Specifying Dynamic Analyses

The ways in which instrumentations and dynamic analyses code are specified and implemented varies widely between languages, toolchains, and application domains. We are particularly concerned with two aspects: the *instrumentation mechanism*—how a base program is modified to gather data—for its effect on performance and compatibility, and the *specification mechanism*—how an analysis defines its instrumentation needs—which greatly affects the complexity of a particular dynamic analyses.

In the domain of JavaScript and web applications, the vast majority of research tools [1, 79, 80, 89, 103, 111, 125] are implemented using source-to-source transformations (also referred to as "source instrumentation") because there is no standardized, cross-platform bytecode for JavaScript. Instrumentation is added by intercepting, parsing, and modifying JavaScript source before it reaches the browser using reverse proxies and JavaScript AST libraries . Source instrumentation is used to install API shims over native methods, add instrumentation hooks around statements or expressions, and interpose on DOM events.

Source instrumentation works for small examples and research prototypes, but its drawbacks are too severe for the approach to be used as a basis for interactive, integrated tools that support program comprehension. Transformed source code is effectively obfuscated, rendering the code unusable by source code editors and debuggers that cannot distinguish application code from instrumentation code. Control flow, allocations, and other dynamic behaviors are also perturbed because instrumentation and application code execute at the same level. Source instrumentation incurs high overheads at instrumentation time and runtime, and is ill-equipped to handle the dynamic features of JavaScript [102] such as `eval` [104] and aliasing of native methods.

In other runtime environments, bytecode instrumentation [6, 55] and dynamic binary translation [7, 86] are the standard mechanisms for modifying programs for analysis purposes. Bytecode instrumentation of JavaScript programs is still relatively uncommon. Though JavaScript bytecode instrumentation is inherently browser-specific and more difficult to implement and maintain, it has significantly better performance and can gather data from the virtual machine that is unavailable

to JavaScript code [8, 102, 104]. Bytecode instrumentation and dynamic binary rewriting can be made compatible with debuggers, profilers, and other tools. For example, Valgrind implements an in-process remote debugging server [99] for GDB which translates debugger commands to work on instrumented code and additionally exposes values of shadow memory and shadow registers.

Dynamic analysis and instrumentation frameworks [6, 7, 75, 86, 111, 125] provide rich APIs which abstract away the significant implementation complexities of instrumentation mechanisms. Generally speaking, frameworks provide a discrete set of instrumentation callbacks (memory read/writes, function call/return, system calls, allocations, etc) or they provide a declarative API for mutating specific AST locations or bytecode sequences. Overscan does not emit events that can be expressed as instrumentation hook pointcuts; other instrumentation events (DOM events, timer registrations, etc.) can be obtained by analyses through built-in instrumentation of the browser platform.

Aspects have been used to declaratively specify instrumentation, especially for coarse-grained analyses in high-level languages such as Java [96, 98, 107] and JavaScript [71, 125]. However, aspects were not designed for instrumentation, so aspect languages have been extended to simplify common tasks like instrumenting basic blocks [28], sharing state across join points [75], and cheaply accessing static and dynamic context at runtime. For greater power, analyses must resort low-level bytecode manipulation libraries [6] to suit their needs. Overscan uses a declarative API for specifying instrumentation hook positions in an AST; the actual data collection and analysis code is implemented by callbacks written in C++. Scry's analyses cannot modify the JavaScript source, and should not perform side-effecting operations such as mutating the JavaScript heap.

Shadow values—duplicated program values that exist in a parallel address space for instrumentation purposes—are a powerful mechanism for implementing complex online dynamic analyses. A dynamic analysis (only one) can add and modify custom annotations in the shadow values as the program accesses the corresponding program values. Valgrind [86] implements shadow registers and shadow memory for x86 binaries. Jalangi [111] implements shadow values for JavaScript by wrapping objects within user-specified segments of JavaScript into a tuple of the application value and the shadow value. Uninstrumented code uses normal application values. ShadowVM [76] is an asynchronous, isolated VM that runs analysis code in a separate thread or process. In addition

to providing shadow values, it provides strong guarantees of isolation and allows an analysis to be profiled and optimized independently of the target program.

2.2.2 Composable and Scoped Dynamic Analyses

Most instrumentation frameworks (with the notable exception of DTrace [16]) are not dynamic: they assume that only one analysis is active at any given time, that the set of active analyses is constant over the program's execution, and that instrumentation is applied equally to all code. Some researchers have investigated ways of making instrumentation more composable, dynamic, and scoped, mainly in the context of Aspects. Ansaloni et al. [4] discuss recent work and open problems in this space, using a running example of three composed dynamic analysis: a calling context profiler, a basic block profiler, and an allocation profiler. We plan to use a similar case study to informally validate the design of Overscan's instrumentation affordances and support for dynamic, scoped instrumentation.

Composable instrumentation requires a framework to mediate the interactions between several competing instrumentations. Clearly, naively instrumenting low-level bytecode or source code is not composable because there's no way to distinguish instrumentation from client code. Researchers of aspect languages have long been concerned with unexpected interplay between aspects that use the same join points, and have developed mechanisms to stratify execution [119, 120] so that aspects cannot advise other aspects. Overscan provides a declarative instrumentation API that multiplexes instrumentation hooks for several active analyses [82], but executes instrumentation code in C++ partially to avoid perturbing execution and introducing meta-levels.

To limit the extent of instrumentation and data collection, prior work relies primarily on static and dynamic context to selectively instrument code or collect data. Overscan's mechanisms for scoping based on time and static scope are inspired by Reflex's support for spatial and temporal filters of behavioral reflection [118], as well as tools that add and remove dynamic instrumentation to running programs [16, 93, 101].

2.2.3 Retroactive and Post-mortem Analysis

Several lines of work have investigated techniques for running a dynamic analysis "offline" by decoupling the analysis from a live execution. Overscan uses the strategy of running dynamic analyses on a replayed execution [20, 110], which we refer to as retroactive analysis. More common in the literature is the strategy of post-hoc trace querying [40, 88, 138] and analysis [69, 96, 97, 98, 100, 102], which we collectively refer to as post-mortem analysis because a live execution is not necessary to perform the analysis. Several projects [7, 21, 97, 111, 138] combine trace-based, query-based and replay-based approaches to achieve interactive response times for common back-in-time queries. The common idea is to save only an index of a program trace's activity, and perform partial replay from a checkpoint to re-materialize a full-fidelity execution trace when necessary.

2.2.4 Visualizing Dynamic Behaviors

Visualization is a large field with many applications to program comprehension. We focus our attention on visualizations of dynamic behaviors, static and dynamic control flow, live execution, and ways in which visualizations are incorporated in development environments and developer workflows.

todo ▶*cite OptimizationCoaching (Racket and JS), LhotakL2004, KoCrystal*◀

todo ▶*cite Orso/Harrold papers that describe some ugly encodings*◀

At the lowest level, many tools visualize dynamic behavior of specific expressions and statements by augmenting source text with overlays, background and foreground color shading [74], context menus [55, 108], inline gutter/scrollbar widgets [74, 108], runtime values [55, 74], or links to other views with information about specific instances [55, 73]. These lightweight visualizations can be used to highlight multi-line units of code, but this can quickly become unmanageable in the presence of namespaces, anonymous event handlers, and other language features that cause definitions and side-effecting statements to be frequently juxtaposed. For example, if two functions are nested in JavaScript, it is unclear whether a statement highlighted in the inner function represents

execution of the inner function or instantiation of the inner function as the outer function execution. Scry handles this situation by using visualizing both notions (Figure **??**) and using hints about the static source structure to explain scoping relationships.

Visualizations of control flow or causality must relate many source elements scattered throughout code that cannot fit into a single source editor view. Graph-oriented visualizations [63] and sequence diagrams [55] are common ways of showing these relationships, but must be used carefully to avoid overwhelming the user with irrelevant information. Like Reacher [63] and WhyLine [55], Scry addresses this challenge by restricting visualization to only those relationships that the user manually expands. Graphs and sequence diagrams are inherently distinct in form from source code, so visualizations must include contextual hints (such as hyperlinks or a call stack) to remind the user of the context of each source element. Scry takes this idea one step further by embedding a source editor into sequence diagrams (Figure **??**) in order to show additional dynamic context around each source element.

An important dimension in visualizing dependencies and relationships is spatial organization: how elements are arranged in space, and how this arrangement implicitly conveys relationships. For example, a timeline of multithreaded execution [127] instantly conveys temporal dependencies among events generated by different threads (even if these are not necessarily accuarate). A call stack visualization [42] can exploit the convention of a stack growing downward to implicitly connote the relationship between callers and callees. Scry uses both of these conventions (along with explicit arrows linking related source elements [55]) to distinguish synchronous caller–callee relationships and asynchronous registration–invocation relationships (Figure **??**). This design is influenced by the VIVIDE programming environment [117], which integrates static and dynamic views of a program on an infinitely horizontally scrolling tape of connected editor windows. Lastly, the Code Canvas line of work [12, 26, 27] explores an infinite two-dimensional pan-and-zoom canvas for arbitrary spatial organization of editors and runtime state. This approach is promising for sharing code investigations with others, but seems difficult to integrate with standard IDE conven-

tions and can become cluttered. The Light Table IDE[1] originally supported arbitrary positioning of editors on a single canvas, but has since reverted back to the dominant tabbed editor interface.

Developers frequently switch among multiple levels of detail and abstraction to better suit their information needs. SHriMP Views [115] were an early exploration of providing multiple levels of detail within the same editor. Other research has used multiple levels of detail to explain cauasality relationships for JavaScript events [1]. Scry primarily exposes multiple levels of detail through progressive data collection. For example, Figure **??** shows several levels of execution profiling. Some visualization tools specifically target discovery of high-level trends over the entire execution [25, 35, 49, 98, 106, 127]. Scry uses low fidelity runtime data from the entire execution to fade out unexecuted code, but relies on the Web Inspector's built-in timeline for discovery of higher-level trends.

2.3 Designing Tools For Program Comprehension

While the technical aspects of implementing low-overhead replay, instrumentation and analyses are challenging and well-studied, their value to a developer ultimately hinges on the effectiveness of the tool with which they interact. Every tool developer hopes that their tool is effective, so why do some tools have a large impact, while others are never used? Researchers in fields such as psychology, sociology, HCI, ergonomics and computer science have developed several theories and models to account for program comprehension and tool use from a cognitive perspective. In this section, we connect these developments to more recent research that focuses on characterizing developers' information needs, and how these questions motivate comprehension tool research.

2.3.1 Cognitive Models of Comprehension and Tool Use

Researchers have long sought to understand the cognitive mechanisms that underly program comprehension and related activities such as debugging. Détienne [29] provides a comprehensive history of cognitive models of program comprehension. Researchers originally modeled program

[1] `https://lighttable.com`

comprehension as a monolithic activity patterned after text comprehension, using concepts such as chunking, top-down and bottom-up comprehension to account for developer's various strategies for reading code. Von Mayrhauser's integrated meta-model [132] is representative of influential cognitive models from the 1980's and early 1990's. Storey [113, 114] provides an insightful catalog of cognitive design elements and design implications for visualization and tool design that arise from these major theories.

In his dissertation, Walenstein [133] adapts the theory of distributed cognition [48] to the domain of software engineering to model exactly how comprehension tools become *useful*. He focuses specifically on the ways in which the cognitive tasks of software development are reconfigured and redistributed by the introduction of developer tools. Using this framing, he proposes to judge usefulness of a developer tool on the basis of how it is able to redistribute cognition between the tool and the developer. For example, by keeping a navigable history of search results, an IDE can offload the significant cognitive effort that would be required for the developer to maintain the same history.

2.3.2 Information Needs and Developers' Questions

In writing about cognitive questions and design elements for software visualizations, Petre et al. [94] raise critical questions about the purpose, design, and interpretation of visualizations. They argue that tool designers must know what programmers actually *do* and ask in practice, so that visualizations and other tools support rather than conflict with these natural representations. Hence, researchers have focused on understanding common modes of developing software [57, 65], collaborating, seeking information [13, 46], describing implicit knowledge [18], and questions during software maintenance [112]. While cognitive models tend to abstract away from detailed examples of information, tool builders necessarily must design for specific use cases and capabilities. Programmers' questions are the crucial link between cognitive models of comprehension and tools that can enhance a programmer's capabilities. Regardless of the specific theory of model of program understanding, all models require information—whether as evidence that tests a hypotheses, as data that solidifies mental models, or as a way to reflect and make explicit the implicit boundary

of what the programmer does and does not know.

In the past 20 years, researchers have shifted from developing large-scale cognitive models and theories to investigating specific aspects of development. While developing the Integrated Meta-Model, von Mayrhauser and Vans [132] began making connections between cognitive models, program understanding tasks and subtasks, and specific information needs formulated as questions. These information needs were gathered from a talk-aloud protocol as part of a study wherein professional developers fixed a bug.

Since von Mayrhauser and Vans's original study, other researchers have used similar study designs to understand developers' practices during code navigation [66, 67] and information-seeking [13, 56, 95, 112], and problem-solving strategies. Most relevant to this dissertation, researchers have catalogued common types of questions, including reachability questions [61, 64], hard-to-answer questions [62] and questions about output and causality [55]. These questions form a comprehensive account of a tool's capabilities from the perspective of its users; many tool papers (including this thesis proposal) begin their motivation by considering how these questions could be answered.

2.4 Working from Visual Output to Runtime States and Code

2.4.1 Linking Visual Output to Program States

Crystal, Whyline

2.4.2 Tracing Program States to Responsible Code

2.4.3 Minimizing and Slicing Dependencies

2.4.4 Visualizing State Differences

Chapter 3

DETERMINISTIC REPLAY FOR WEB PROGRAMS

3.1 Motivation

Debugging is often an iterative process in which developers repeatedly adjust their view on a program's execution by adding and removing breakpoints and inspecting program state at different times and locations. This iteration requires a developer to repeatedly reproduce the behavior they are inspecting, which can be time-consuming and error-prone [**?**]. In the case of interactive programs, even reproducing a failure can be difficult or impossible: failures can occur on mouse drag events, be time-dependent, or simply occur too infrequently for a developer to easily reach a program state suitable for debugging.

Deterministic record/replay [23] is a technique that can be used to record a behavior once and then deterministically replay it repeatedly and automatically, without user interaction. Though it seems that the capability to record and replay executions should be useful for debugging, no prior work has described actual use cases for these capabilities, or how to best expose these capabilities via user interface affordances. At most, prior systems demonstrate feasibility by providing a simple VCR-like interface [80, 131] that can record and replay linearly. Many record/replay systems have no UI, and are controlled via commands to the debugger or special APIs [43, 53, 109]. Finally, prior work does not consider how record/replay capabilities can interoperate with and enhance existing debugging tools like breakpoints and logging. Prior tool instantiations [3, 80, 103] are inappropriate for debugging because they can interfere with program performance, determinism, and breakpoint use, and they are difficult to deploy.

This paper presents Timelapse, a developer tool for capturing and replaying web application behaviors during debugging, and Dolos, a novel record/replay infrastructure for web applications. A developer can use Timelapse's interactive visualizations of program inputs to find and seek

non-linearly through the recording, and then use the debugger or other tools to understand program behavior. To ensure deterministic execution, Dolos captures and reuses user input, network responses, and other nondeterministic inputs as the program executes. It does this in a purely additive way—without impeding the use of other tools such as debuggers—via a novel adaptation of virtual machine record/replay techniques to the execution environment of web browsers. Debugging tools are particularly important for interactive web applications because the web's highly dynamic, event-driven programming model stymies traditional static program analysis techniques.

This paper makes the following contributions:

- Dolos: a fast, scalable, precise, and practical infrastructure for deterministic record/replay of web applications.
- Timelapse: a developer tool for creating, visualizing, and navigating program recordings during debugging tasks.
- The first user study to explore how and when developers use record/replay tools during realistic debugging tasks.

Dolos and Timelapse are free software: our source code, study materials, and data are available on the project website[1].

We first discuss the design of Timelapse, and use a scenario to illustrate how Timelapse supports recording, reproducing, and debugging interactive behaviors. Next, we present the design and implementation of the underlying Dolos record/replay infrastructure. We discuss the results of a small study to see how developers use Timelapse in open-ended debugging tasks. Finally, we conclude with related work and present several implications for future interactive debugging tools.

3.2 Record/replay Infrastructure

Dolos is the underlying record/replay infrastructure that enables the Timelapse tools and user interfaces. For Timelapse to be useful during debugging tasks, Dolos must not disrupt existing

[1] `http://github.com/burg/timelapse/`

developer tools and workflows. Concretely, this entails the following four design goals:

1. **Low overhead.** Recording must introduce minimal performance overhead, because many interactive web applications are performance-sensitive. Replaying must be fast so that users can quickly navigate through recordings.

2. **Deterministic replay.** Recordings must exactly reproduce observable program behavior when replaying. Replaying should not have external effects.

3. **Non-interference.** Recording and replaying must not interfere with the use of tools such as breakpoints, watchpoints, profilers, element inspectors, and logging.

4. **Deployability.** Recording and replaying must require no special installation, configuration, or elevated user privileges.

To the best of our knowledge, no prior record/replay tool satisfies all of these constraints. Source-level instrumentation of JavaScript [78] can perturb performance [103] and relies on hard-to-deploy[2] proxies to perform source rewriting. Rewriting techniques can interfere with a developer's expectations by obfuscating source code or by causing the debugger to unexpectedly interact with code the developer did not write. todo ▶*reference related work section on how to instrument code.*◀ User-space record/replay library tools [80, 109] execute in the same address space as the target program and use wrappers to interpose on nondeterministic APIs (such as the `Date` constructor). These tools are inherently incompatible with breakpoint debuggers: because they piggyback on the target program, pausing at a breakpoint will also prevent the record/replay library's mechanisms from executing. Virtual machine replay debugging [131] ensures deterministic execution of the entire browser instead of one page context—preventing the use of built-in debugging tools [**?** **?**], which cannot run independently from other browser components. Macro-like replay tools such as Selenium WebDriver [**?**], CoScripter [72], or Sikuli [137] are commonly used for workflow

[2]Proxies used for instrumentation and debugging like Fiddler [122] require elevated privileges, manual configuration, and intentional man-in-the-middle attacks to subvert SSL encryption [121].

automation; towards that end, they only record and replay user input, and not other program inputs such as network traffic, dates and times, and other sources of non-determinism. Replaying only user input is not deterministic enough to consistently reproduce web application behaviors. These external tools are unaware of internal system state or external server state: for example, they may try to provide input while the program is paused at a breakpoint, or cause the program to ask a server for resources that are no longer valid.

3.2.1 Web Browser Architecture

We use the term *web interpreter* to refer to the interpreter-like core of web browsers. The web interpreter processes inputs from the network, user, and timers; executes JavaScript; computes page layout; and renders output. We do not consider features that do not affect the results of computation, such as bookmarks and a browser's address bar. The web interpreter schedules asynchronous computation using a cooperative single-threaded event loop. It communicates with embedders (Safari, Google Chrome) and platforms (Linux, Mac OS X) via embedder and platform APIs (Figure 3.1). Both APIs are bidirectional: the web interpreter can call "out" to collect environmental information from the platfrom or delegate security policies to the embedder; the platform and embedder can call "in" to schedule computation (asynchronous timers, user input, network traffic) in the web interpreter's event loop.

3.2.2 Design

Our requirements of low overhead, deterministic replay, non-interference, and deployability are closest to the design goals of virtual machine-based record/replay systems. Well-known systems in this space, such as ReVirt [33] and VMWare Replay Debugging [131], achieve deterministic record/replay via the "instruction-counting trick". A hypervisor interposes on nondeterministic machine instructions, but executes ordinary instructions natively on the bare-metal processor. Interrupts are captured and injected accurately by counting their position in the instruction stream. This technique satisfies our four design goals: it is fast, ensures precise replay, is compatible with

debuggers, and requires minimal modifications to the runtime environment.

The execution model of web programs does not have machine instructions or interrupts, but there are parallels between virtual machines and web browsers. The Dolos infrastructure makes the following novel substitutions to make the "instruction-counting trick" work for the web's execution model:

- Instead of simulating deterministic hardware, Dolos simulates deterministic responses from embedder and platform when the web interpreter calls out to them.

- Instead of hardware interrupts, Dolos captures and simulates the subset of inbound embedder and platform API calls that trigger computation within the web application.

- Instead of counting instructions as a unit of execution progress, Dolos counts DOM event dispatches—precursors to the execution of JavaScript code—that may have deterministic or nondeterministic effects.

3.2.3 Implementation

The Dolos infrastructure is a modified version of the WebKit[3] browser toolkit. We chose WebKit because it contains the most widely-deployed web interpreter (WebCore) and developer toolchain (Web Inspector). Our deployment model is straightforward: a version of WebKit built with Timelapse and Dolos support is distributed as a dynamic library, and can be used with existing WebKit embedders such as Safari or Google Chrome by adjusting the dynamic library load path.

Recording and Replaying

Hypervisors (and record/replay systems based on them) are fast in part because most execution occurs at full speed on bare hardware. Similarly, Dolos achieves high record/replay performance

[3] `http://www.webkit.org/`

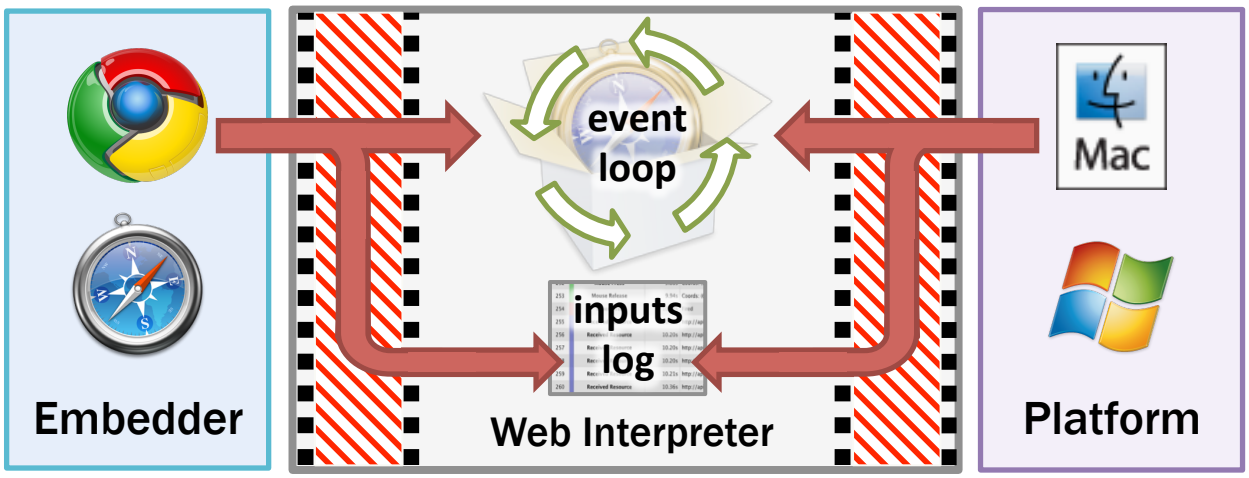

Figure 3.1: The flow of inputs while recording execution. Boxes delineate module boundaries. Arrows indicate the flow of program inputs. The web interpreter receives input and sends output via embedder and platform APIs (thick lines). During recording, input shims (striped regions) record program inputs delivered via the public APIs, and pass on the inputs to the unmodified event loop.

in part by using the same code paths as normal execution when recording and replaying program inputs. Dolos captures and replays calls to the embedder and platform APIs using *shims*—thin layers used to observe external API calls or simulate external API calls from within. During recording (Figure 3.1), the shims record all API calls that affect program determinism. During replay, these API calls are simulated (Figure 3.2) to deterministically reproduce the recorded program behavior. To prevent user interactions from causing a different execution during replay, the shims block external API calls while allowing simulated calls to proceed.

During recording, Dolos saves the timing, ordering, and details of inbound calls to the web interpreter that are pertinent to the recorded web program's execution. During replay, these inbound calls are simulated by Dolos, rather than actually being issued by the embedder or platform. Dolos's approach to capturing and replaying computation-triggering calls is applicable to programs implemented using an event-loop-based UI framework. Dolos can seek to and/or pause execution

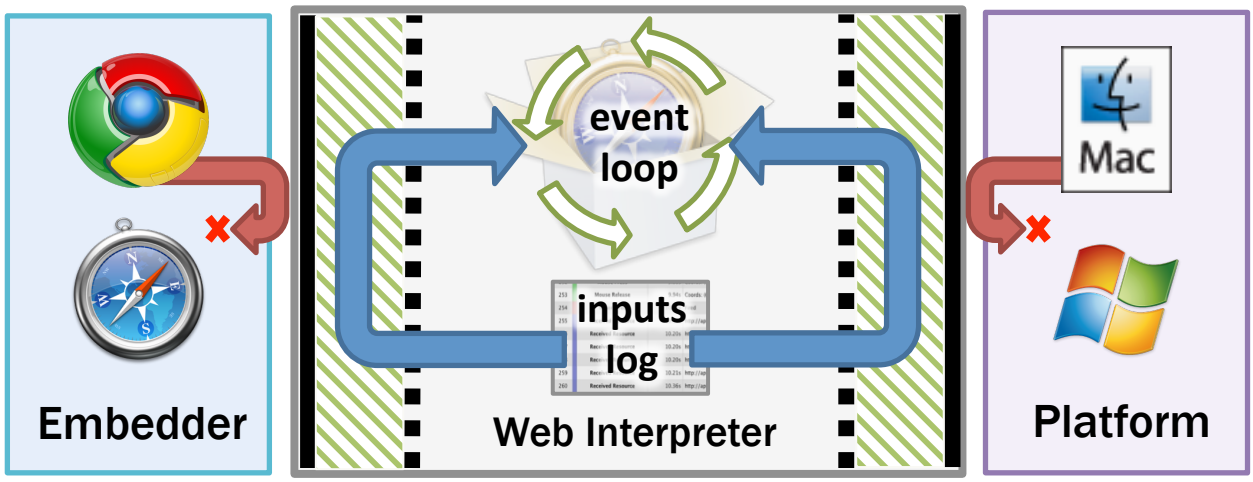

Figure 3.2: The flow of inputs while replaying execution. During replay, the interpreter's replay controller uses the input log to simulate the sequence of recorded public API calls into the interpreter (blue arrows). External calls to public APIs are ignored (solid black line) to prevent user interactions from causing a different execution. Calls from the web interpreter to platform APIs (i.e., to get the current time) use memoized return values and do not have external effects.

prior to any simulated inbound call. Because the event loop is transparent to web programs, Dolos "pauses" execution without using breakpoints by simply not simulating the next computation-triggering inbound call. One can use breakpoints to pause JavaScript execution at any statement, but cannot use them to pause execution before the web interpreter interprets an inbound call.

Program inputs

Like hypervisor record/replay systems, Dolos classifies inputs to the web interpreter—that is, relevant inbound and outbound embedder and platform API calls—as either interrupts or environmental nondeterminism. Table 3.1 enumerates the major sources of interrupt-like inputs and environmental inputs. Conceptually, interrupts cause computation to happen, while environmental inputs embody nondeterministic aspects of the environment. Interrupts are API calls *from* the embedder or platform to the interpreter; environmental inputs are API calls *to* the embedder or platform from

Input	Classification	DOM Derivatives
Keyboard strokes	Interrupt	keyboard events
Mouse input	Interrupt	mouse events
Scroll wheel	Interrupt	scroll events
Page focus/blur	Interrupt	focus, blur
Window resize	Interrupt	resize
Document navigation	Interrupt	unload, pagehide
Timer callbacks	Interrupt	(none)
Asynchronous events	Interrupt	animation events
Network response	Interrupt	AJAX, images, data
Random numbers	Environment	Math.random
Browser properties	Environment	window.navigator
Current time	Environment	Date.now
Resource cache	Environment	(none)
Persistent state	Environment	cookies, local storage

Table 3.1: Major classes of input as defined by Dolos.

the interpreter.

Dolos records and replays embedder/platform API calls rather than individual DOM events [80, 91, 103] which result from the web interpreter's interpretation of inbound API calls. For example, instead of recording individual DOM `click` events, Dolos captures the embedder's calls to the web interpreter's `handleMousePress` API. This in turn may trigger one, many, or no DOM event dispatches, depending on the interpreter's state. Dolos's strategy reduces log size, runtime overhead, and implementation complexity because it does not require event targets to be serialized when recording or re-targeted when replaying.[4]

Dolos captures and replays network traffic in the same way that other event loop callbacks are

[4]Guo et al. [43] report on the space and time benefits of memoizing application-level API calls instead of low-level system calls [109].

Program			Run Time					
Name	**Description**	**Workload**	**Bottleneck**	**Baseline**	**Disabled**	**Recording**	**Replaying**	**Seeking**	
JSLinux	x86 emulator	Run until login	network	10.5s	1.00×	1.65×	1.65×	0.37×	24
JS Raytracer	ray-tracer	Complete run	CPU	6.3s	1.00×	1.01×	1.17×	1.02×	24.
Space Invaders	video game	Scripted gameplay	timers	25.8s	1.00×	1.03×	1.22×	0.25×	24
Mozilla.org	home page	Read latest news	user	22.3s	1.00×	1.00×	1.09×	0.23×	18
CodeMirror	text editor	Edit a document	user	16.6s	1.00×	1.00×	1.03×	0.07×	57
Colorpicker	jQuery widget	Reproduce defect	user	15.3s	1.00×	1.00×	1.07×	0.13×	11
DuckDuckGo	search engine	Browse results	user	14.1s	1.00×	1.00×	1.08×	0.19×	11

Table 3.2: Overhead for three non-interactive and four interactive programs. "Baseline" is unmodified WebKit, and "Disabled" is Dolos when neither record nor replay is enabled. Log size is given for the in-memory representation, the uncompressed log file, and the compressed log file. Site content is images, scripts, and HTML.

handled. When capturing, Dolos caches the HTTP headers and raw data of network responses as they are passed from the embedder to the web interpreter. When replaying, Dolos blocks outbound network requests from actually reaching the network. Dolos simulates deterministic network responses by reusing cached HTTP headers and data. For example, when loading images asynchronously on Flickr, Dolos caches all image data when capturing. When replaying, Dolos simulates network responses by reusing cached image data, and never communicates with Flickr servers.

For environmental inputs, Dolos uses shims to memoize the web interpreter's C++ calls to the platform or embedder APIs rather than memoizing calls to nondeterministic JavaScript functions. For example, Dolos does not record the return value of JavaScript's `Date.now()` function; instead, Dolos memoizes the JavaScript engine's calls to the `currentTimeMS()` platform API inside the implementation of `Date.now()`.

Replay fidelity

The Dolos infrastructure guarantees identical execution when recording and replaying, up to the order and contents of DOM events dispatched. JavaScript execution is completely deterministic. There is no guarantee regarding number of layout reflows or paints due to time compression or internal browser nondeterminism. This is unimportant because visual output cannot affect the determinism of JavaScript computation: JavaScript client code can ask for computed layout data, but will block until the layout algorithm runs on the current version of the DOM tree. Rendering and painting activity is not exposed to client code at all.

To the best of our knowledge, Dolos is the first web record/replay infrastructure that *detects* execution divergence. Timelapse warns users when divergences are detected or when known-nondeterministic APIs are used, and allows them to abort replay, ignore divergences, or report feature requests to the Dolos developers. Dolos uses differences in measures such as DOM node counts and event dispatch counts to detect unexpected execution divergences caused by bugs in Dolos itself. This has helped us find and address obscure sources of nondeterminism, such as resource caching effects, asynchrony in the HTML parser, implicit window size and focus state, and improper multiplexing of inputs to nested `iframe` elements.

The Dolos prototype does not address all known sources of nondeterminism, such as the Touch, Battery, Sensor, Screen, or Clipboard APIs, among others. There are no conceptual barriers to supporting these features: they are implemented in terms of standardized DOM events and interfaces, making them relatively easy to interpose upon using techniques described in this paper. Each new program input requires local changes to route control flow through a shim, plus a helper class to serialize and simulate the program input.

Engineering cost

Dolos's design scales to new platforms, embedders, and sources of nondeterminism because inputs are recorded and replayed at existing module boundaries. Implementing Dolos required minimal changes to WebKit's architecture: namely, the insertion of shims just beneath the web interpreter's

public embedder and platform APIs (shown in Figure 3.1 and in Figure 3.2). The Dolos infrastructure consists of 7.6K SLOC (74 new files, 75 modified files). For comparison, WebKit contains about 1.38M SLOC.

3.2.4 Overhead

Dolos's record/replay performance slowdown is unnoticeable ($< 1.1\times$) for interactive workloads and modest ($\leq 1.65\times$) for non-interactive benchmarks without any significant optimization efforts (Table 3.2). Recording overhead and the amount of data collected scales with user, network, and nondeterministic inputs, not CPU time.

We report the geometric mean of 10 runs (except for interactive runs, which were recorded once but replayed 10 times). The standard deviation was always less than 10% of the geometric mean. We cleared network resource caches between executions to avoid memory and disk cache nondeterminism. In measurements of execution time, We used local copies of benchmarks to avoid network nondeterminism.

Recording has almost no time overhead: execution times are dominated by the subject program. Replaying at $1\times$ speed is marginally slower than a normal execution due to extra work performed by Dolos, and seeking (fast replaying) is much faster because it elides user and network waits from the recorded execution.

While being created or replayed, recordings are stored in-memory and consume modest amounts of memory (first column in the data size section of Table 3.2). When serialized, the recordings are highly compressible. A recording's length is limited only by main memory; in our experience, users attempt to minimize recording length to reduce the number of irrelevant inputs. Timelapse's linear presentation of time does not scale well to long executions. This could be ameliorated by using nonlinear time scaling techniques [58, 127].

3.2.5 Limitations

Dolos only ensures deterministic execution for *client-side* portions of web applications; it records and simulates client interactions with a remote server, but does not directly capture server-side state. Tools that link client- and server-side execution traces [**?**] may benefit from the additional runtime context provided by a Dolos recording.

Dolos cannot control the determinism of local, external software components such as Flash, Silverlight, or other plugins. However, plugins interact via the embedder API; Dolos could handle plugin nondeterminism in the same way that other embedder nondeterminism is handled.

Web interpreters provide many API calls that do not affect program determinism. For example, WebKit's web interpreter includes APIs for usability features like native spell-checking, in-page search, and accessibility. Dolos does not record or replay these API calls because a developer may wish to use such features differently during replay, and these features do not affect the determinism of the web program.

Dolos's hypervisor-like record/replay strategy relies on the layered architecture of WebKit. It is not directly applicable to systems without clear API boundaries between the embedder, platform, and web interpreter. For example, the Gecko web interpreter used by Firefox is implemented by dozens of decoupled components rather than a monolithic module. This design makes it easy to extend the browser, but difficult to record and replay only the subset of components that affect the specific web program's execution, as opposed to those are used by browser features, browser extensions, or several web programs at once. Shims for Gecko would have the same behavior as Dolos's shims, but they would be placed throughout the web interpreter rather than at coarse abstraction boundaries.

3.2.6 Visualizing and Exploring Recordings and Traces

todo ▶*merge into related work section*◀ In contrast to the dearth of interactive record/replay tools, there have been many tools [55, 98] to visualize, navigate, and explore execution traces[5] generated by logging instrumentation. This is because execution traces are several orders of magnitude larger and contain very low-level details: the only way to understand them is to use elaborate search, analysis, and visualization tools. While Timelapse visualizes an execution as the temporal ordering of its inputs, trace-based debugging tools [55, 127] infer and display higher-level structural or causal relationships observed during execution. Timelapse's affordances primarily support navigation through the recording with respect to program inputs, while trace-based tools focus directly on aids to program comprehension (such as supporting causal inference or answering questions about what happened [55]).

Unfortunately, the practicality of many trace-based debugging tools is limited by the performance of modern hardware and the size of generated execution traces. Modern profilers, logging, and lightweight instrumentation systems [16] (and visualization tools [127] built on top of them) have acceptable performance because they capture infrequent high-level data or perform sampling. In contrast, heavyweight fine-grained execution trace collection introduces up to an order of magnitude slowdown [55, 88]. Generated traces and their indices [97, 98] are very large and often limited by the size of main memory.

3.2.7 Supporting Behavior Reproduction and Dissemination

To the best of our knowledge, deterministic record/replay systems that support dissemination of behaviors have only been widely deployed as part of video game engines [30]. Recordings of gameplay are artifacts shared between users for entertainment and education. These recordings are also a critical tool for debugging video game engines and their network protocols [123]. In the wider software development community, bug reporting systems [39] and practices [**?**] emphasize

[5] *Execution traces* consist of intermediate program states logged over time, while Dolos's recordings consist only of the program's inputs.

the sharing of evidence such as program output (e.g., screenshots, stack traces, logs, memory dumps) and program input (e.g, test cases, configurations, and files). Developers investigate bug reports with user-written reproduction steps.

While we have focused on the utility of record/replay systems for debugging, such systems are also useful for creating and evaluating software. Prior work has used record/replay of real captured data to provide a consistent, interactive means for prototyping sensor processing [17, 87] and computer vision [52] algorithms. More generally, macro-replay systems for reproducing user [**?**] and network [122] input are used for prototyping and testing web applications and other user interfaces. Dolos recordings contain a superset of these inputs; it is possible to synthesize a macro (i.e, automated test case) for use with other tools. The JSBench tool [103] uses this strategy to synthesize standalone web benchmarks. Derived inputs may improve the results of state-exploration tools such as Crawljax [79] by providing real, captured input traces.

3.3 Conclusion and Future Work

todo ►*convert to a discussion section, move other stuff.*◄ Together, Timelapse and Dolos constitute the first toolchain designed for interactively capturing and replaying web application behaviors during debugging. Timelapse focuses on browsing, visualizing, and navigating program states to support behavior reproduction during debugging tasks. Our user study confirmed that behavior reproduction was a significant activity in realistic debugging tasks, and Timelapse assisted some developers in locating and automatically reproducing behaviors of interest. The Dolos infrastructure uses a novel adaptation of instruction-counting record/replay techniques to reproduce web application behaviors. Our prototype demonstrates that deterministic record/replay can be implemented within browsers in an additive way—without impacting performance or determinism, impeding tool use, or requiring configuration—and is a platform for new debugging aids.

Prior work assumes that executions are in short supply during debugging, and that developers know a priori what sorts of analysis and data they want before reproducing behavior. In future work, we want to disrupt this status quo. On-demand replay (in the foreground, background, or offline) could change the feasibility of useful program understanding tools [55] or dynamic analy-

ses [20] that, heretofore, have been considered too expensive for practical (always-on) use. Using the Dolos infrastructure, we intend to transform prior works in dynamic analysis and trace visualization into on-demand, interactive tools that a developer can quickly employ when necessary. We believe that when combined with on-demand replay, *post mortem* trace visualization and program understanding tools will become *in vivo* tools for understanding program behavior at runtime.

Chapter 4

LOGGING AND NAVIGATING TO PAST PROGRAM STATES

4.1 Motivation

Reproducing and inspecting specific program states is a fundamental task during activities such as debugging and reverse-engineering. This task is challenging for experts and novices because tools like breakpoints and logging only indirectly support it. When properly placed, logging statements can alert a programmer to relevant runtime states, but logging statements cannot suspend the program for further inspection. Conversely, breakpoint debuggers can suspend a program and control its execution at a fine-grained level, but can only suspend control flow at *future* times rather than at past program outputs. Because of these limitations, breakpoints and logging are subject to a large gulf of execution, making it difficult to predict whether any particular tool will be helpful for the task at hand.

Prior work has investigated deterministic replay techniques—which operate by controlling sources of nondeterminism at runtime—as a basis for automatically reproducing specific executions. Dolos and similar replay infrastructures [80] todo ▶*addref to related work section*◀ only provide affordances for navigating a captured execution by its user inputs, event loop tasks, or low-level signals. To suspend execution at a specific program point, a developer must isolate and replay to the preceding input and then use a breakpoint debugger to drive execution to a specific statement.

In prior work, researchers have observed [15, 56, 105] that developers greatly prefer to navigate executions by *outputs*, rather than by inputs. A program's outputs can correspond to relevant program states, and developers often work backwards from outputs when attempting to understand runtime behavior [55]. To this end, we contribute two extensions to previous replay systems that realize output-oriented navigation: *time-indexed outputs* and *data probes*.

Time-indexed outputs empower a developer to see the logged output they want to investigate and, with a single click, jump to the exact program statement that produced the output. The algorithm for seeking to time-indexed outputs is simple and incurs little overhead. By reducing the task of reproducing program states to only require a single click, time-indexed outputs make it possible for a developer to easily navigate between task-relevant instants of execution without disruptive context switches.

Our second contribution is *data probes*, a feature that allows a developer to retroactively add logging statements to a captured execution without editing program text and without re-executing the program. A data probe may have multiple *probe expressions* that are evaluated and logged to produce new time-indexed outputs. Like a breakpoint, a probe is placed at a single statement in the program; when the statement executes, the probe's expressions are evaluated to create *probe samples*. Data probes and probe samples are saved across multiple playbacks of a captured execution. Using data probes, a programmer can interactively discover, compare, and navigate to interesting program states in the past or the future without excessive planning or manual effort.

The rest of this paper explains these two contributions and describes a prototype[1] based on the WebKit browser toolkit.

4.2 An Example

Time-indexed outputs and data probes are designed to automate the tedious, error-prone tasks of suspending execution and logging specific program states within a captured recording. This section uses a program maintenance task to demonstrate advantageous uses of probes and time-indexed outputs.

Color Picker[2] is a jQuery plugin that implements a color picker widget for RGB and HSV colorspaces. Karla, a developer, uses the widget in her web application. She is investigating a bug that manifests when a user manipulates the color picker's color component sliders (Figure 4.2). Each slider should adjust the value of one red, green, or blue (RGB) component independently of the

[1] `http://bit.ly/1yi0gOo` (Obfuscated for blind review)

other two components, but sometimes moving one slider incorrectly affects more than one component. The bug is caused by unnecessary rounding in the algorithm that converts values between RGB and HSV color spaces. Understanding and fixing this bug is difficult for several reasons: reproducing the bug requires manual user interaction; the wrong results appear only sporadically and are not persistent; and it is hard to isolate and investigate specific computations, such as a single event handler execution.

Karla starts by using a record/replay tool (such as Dolos [15]) to capture a recording that demonstrates the steps to reproduce the Color Picker bug. This recording makes further reproduction simple by allowing Karla to quickly jump to the input just prior to the failure, but she still must explore the thousands of lines of code that execute after this input using traditional debugging tools. She still must set breakpoints to suspend execution at specific lines of code, which is tedious because the program must be re-executed to test whether the breakpoints were positioned correctly. Logging program states is similarly laborious: to log color component changes, Karla has to edit the widget's source code, rebuild and re-deploy the website, capture a new recording, and then view the logged outputs.

Data probes and time-indexed outputs simplify Karla's investigation of the buggy interaction. To create a foothold for observing runtime behavior, Karla uses data probes to log RGB values as they change, since past component values are not stored or logged to the console. After using the built-in timelines view (Figure 4.1.a) to see what code executed during the recording, she guesses that the `moveIncrement` handler may contain the RGB values she wants to log. To see the effects of each drag event, Karla adds data probes before and after the handler modifies color components (at `colorpicker.js:127` and `colorpicker.js:131`, as seen in Figure 4.1.b). These data probes capture the value of the expressions `color.r`, `color.g`, and `color.b` (Figure 4.1.c) each time the associated line executes. Karla then replays the recording again to generate new probe samples. As the recording is replayed, the probe sidebar (Figure 4.1.c) begins to populate with probe samples. Looking at temporally-ordered probe samples in the console (Figure 4.3), Karla quickly sees a few

[2] `http://www.eyecon.ro/colorpicker/`

instances where multiple components changed in a single drag event.

To better understand what happened, Karla wants to inspect the program states leading up to a suspicious probe sample. Since all probe samples are also time-indexed outputs, Karla can suspend execution immediately before the offending drag event handler by double-clicking on a probe sample collected at that time (Figure 4.1.c and Figure 4.3). From that instant of execution, she can use the debugger to step into the event handler and work towards the root cause with the aid of actual runtime values. With time-indexed outputs and data probes, she's likely able to do this much faster and more systematically than with breakpoints and logging alone.

4.3 Implementation

Our prototype of data probes and time-indexed output is built on top of the Dolos infrastructure. Dolos, data probes, and time-indexed outputs have been incorporated into the open-source WebKit browser toolkit[3].

4.3.1 Creating and Evaluating Data Probes

The goal of data probes is to add temporary logging expressions to a program as it is running. Like breakpoints, probes are associated with a specific source statement and are processed just prior to the statement's execution. In fact, our prototype implements probes as a special breakpoint action that evaluates the probe's expressions and saves the results. Probe expressions can capture a wide range of values, including scalars such as numbers or strings, or non-scalar values, such as arrays, objects, or in the case of the web, DOM elements.

The probes user interface supports comparing, relating, and navigating to probe samples. The probes sidebar (Figure 4.1.c) persists collected samples for all subsequent playbacks of the recording. This allows a user to stitch together a map of probe samples across the whole recording without necessarily replaying it contiguously with a specific set of data probes. The probes sidebar groups probe samples by call site to support comparisons. Probe samples are also printed to the

[3] `https://www.webkit.org`

console in execution order to provide a navigable, time-synchronized log. During playback, the console shows output produced up to the current instant, but not later outputs captured in a previous playback.

4.3.2 Replaying to Output-Producing Statements

The goal of time-indexed outputs is to suspend execution at the instant when a statement produces a specific output.[4] In a deterministic record/replay setting, output-producing statement evaluations can be uniquely indexed by associating a counter with each such statement. The counter increments when its statement executes, and the produced output is tagged with the current counter value. To suspend execution at the statement that produced time-indexed output n, a naive approach is to set a breakpoint at the statement, re-execute, and resume from the breakpoint $n-1$ times. However, this approach is too slow and brittle for realistic use: since counter values are relative to the beginning of the recording, a full replay is required to suspend execution at output-producing statements or tag probe samples from a new data probe.

We avoid these drawbacks by making counter values relative to the currently executing event loop task. In a deterministic record/replay setting, event loop tasks that cause JavaScript to execute—such as timer callbacks, network callbacks, or user inputs—are always executed in the same order. If a replay infrastructure can uniquely refer to specific event loop tasks, then an evaluation of an output-producing statement can be uniquely indexed by the statement's counter value and the preceding event loop task. This optimization has several important implications:

- We can revisit time-indexed outputs without restarting playback from the beginning of a recording.

- We accumulate partial knowledge of time-indexed outputs across multiple discontiguous playbacks.

---

[4] We assume that tools can automatically identify output-producing statements. Our prototype tags outputs from data probes and `console.log` statements.

- We can automatically discover new probe samples (i.e., time-indexed outputs) in unknown sections of a recording.

Our prototype adds counter values by modifying the implementations of `console.log` and data probes to tag outputs that these statements produce. The prototype seeks to a specific time-indexed output in three phases: first, it uses Dolos to replay up to the preceding event loop task; second, a hidden breakpoint is added at the output-producing statement; third, the debugger pauses and resumes $n - 1$ times from the hidden breakpoint; and last, execution is suspended a final time immediately before the desired evaluation.

4.3.3 Minimizing Breakpoint Use

While breakpoints are a useful mechanism for collecting probe samples and replaying to time-indexed outputs, their use incurs a significant ($10\times$) performance overhead.[5] This slowdown can negate the interactive qualities of data probes and time-indexed outputs, which may lead a developer to use them in a more cumbersome batch-oriented manner. Our prototype uses several strategies to minimize the use of breakpoints. When replaying to a time-indexed output, it completely disables the debugger until the preceding event loop task, and then sets a single breakpoint at the output-producing statement. This limits breakpoint-induced slowdowns to a single event loop turn. Our prototype also optimizes its use of breakpoints when sampling data probes. To preserve the developer tool's ability to interactively inspect complex JavaScript values, data probes must be resampled on subsequent playbacks. However, if a probe sample can be serialized and reused without a live object reference, then resampling can be avoided.

4.4 Related Work

Several lines of research investigate ways of tightening the feedback loop of editing code, looking at output, and debugging further. This section discusses those which use program output as beacons

[5]The mere presence of breakpoints deoptimizes emitted bytecode and prevents most adaptive optimizations such as inline caches.

for navigating through an execution, and those which simplify the process of gathering runtime state.

4.4.1 Capturing and navigating executions

Two approaches—deterministic record/replay and post-mortem trace analysis—dominate research into capturing and navigating through executions. Deterministic replay research traditionally focuses on capturing executions with low overhead and high fidelity. Recent prototypes such as Aftersight [20] and Jalangi [111] perform dynamic analysis on-demand during replayed executions. Data probes also gather data from a replayed execution, but their placement is driven by user interaction rather than pre-defined analyses.

Tools based on *post-mortem trace analyses* save all program operations into a large trace file, and later query the trace to reconstruct intermediate program states or program output. These systems incur 1–2 orders of magnitude slowdown during recording and amortize that cost by never re-executing the program. In practice, only deterministic record/replay tools have low enough overheads to be used for interactive programs.

The Whyline for Java [55] is a heavyweight trace analysis tool that allows developers to ask why- and why-not questions about program output. It can reconstruct a program's visual output, textual output, and intermediate program state from the operation trace, and it uses program slicing techniques over the trace to answer program understanding questions on demand. By leveraging deterministic replay instead of post-mortem trace analyses, time-indexed outputs provides many of the same affordances as Whyline, but with near-zero runtime overhead and at interactive speeds on real web applications.

4.4.2 Live programming systems

The live programming paradigm [44] emphasizes tight feedback loops, typically by blurring or removing delays between editing a program and seeing effects of the changes. Live programming tools support quick iteration on inputs or the program itself, and often eschew imperative pro-

gramming models to better support these goals. Below, we discuss two analogues to data probes and time-indexed outputs designed for other programming environments. While the affordances offered are similar, only data probes and time-indexed outputs are designed for the mainstream domain of imperative, interactive programs such as web applications.

DejaVu [52] supports interactive debugging and development of image processing algorithms. Similar to data probes, a user can add new debugging outputs (e.g. image filters, inferred skeletal models) which are automatically computed and juxtaposed with prior output on a timeline. Using the output timeline, the user can revert execution to a specific input or rendered output frame and inspect the program's state. Time-indexed outputs support a similar interaction for textual outputs, such as console logging or probe samples.

YingYang [77] is a programming environment that integrates live execution feedback with an emphasis on *traces* (similar to console output) and *probes* (similar to our probes). It supports rewinding execution to a logged output, and can substitute concrete values into the code/stack frame to explain the output's derivation. YingYang is built atop the Glitch live programming runtime. Glitch incrementally repairs program state as inputs or the program itself changes, so it has no notion of state over time, nor does it support temporally-ordered inputs. In contrast, our data probes and time-indexed output can be used to navigate recordings of imperative, interactive programs.

4.5 Future Work

Data probes and time-indexed outputs are a step towards the Whyline [55] vision of interactively investigating runtime behaviors. Data probes and time-indexed output provide a way to navigate captured recordings via console outputs and intermediate script states. In future work, we will continue to explore the Whyline vision while maintaining interactive performance and integration with production web development tools. We want to improve links between visual output and the code fragments that produced the output. We also plan to expand the scope of probes to include other program states, such as changes to the DOM tree structure or changes to an element's appearance or position.

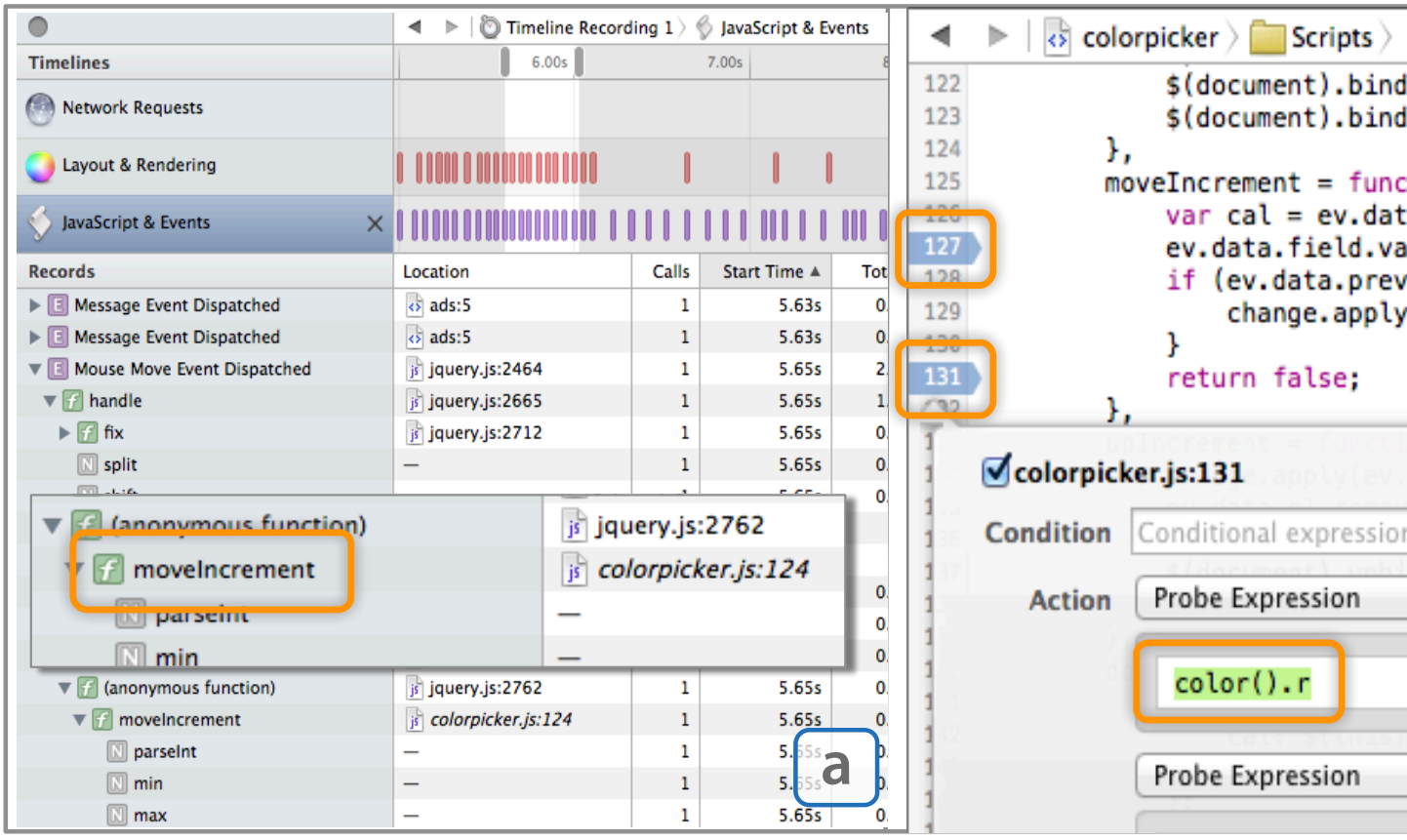

Figure 4.1: Using data probes to revert execution to relevant program states. In (a), Karla inspects the captured recording to find the `moveIncrement` drag event handler (highlighted). In (b), she adds two *data probes* (at lines 127 and 131) to log how color component values changed. In (c), she reverts execution directly to a probe sample (top, selected row) that immediately precedes erroneous component values (below, highlighted).

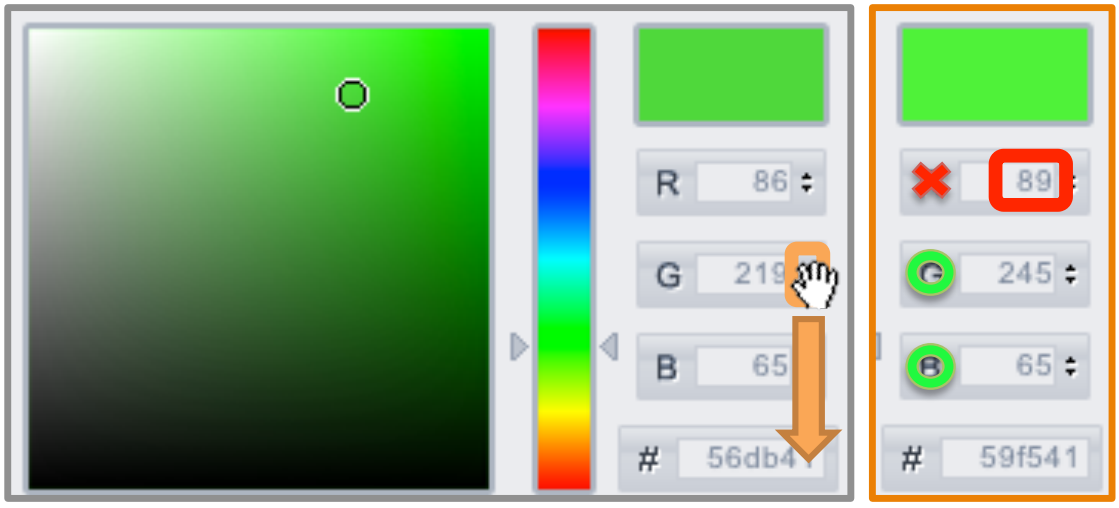

Figure 4.2: The Color Picker widget. When the G color component is adjusted downward (left), the R component unexpectedly changes (right).

```
[colorpicker.js:127] color.r 255 color.g 49 color.b 13
[colorpicker.js:131] color.r 255 color.g 49 color.b 13
[colorpicker.js:127] color.r 255 color.g 49 color.b 13
[colorpicker.js:131] color.r 255 color.g 50 color.b 14
[colorpicker.js:127] color.r 255 color.g 50 color.b 14
[colorpicker.js:131] color.r 255 color.g 51 color.b 15
```

Figure 4.3: Probe samples ordered temporally in console output. The selected row (green) shows both G and B components changing at the same time. Karla double-clicks on the preceding probe sample to suspend execution prior to the faulty event handler's execution.

Chapter 5

A USER INTERFACE FOR CAPTURING AND REPLAYING EXECUTIONS

5.1 Motivation

todo ▶*Fill this in, based on assumptions from prior work and shortcomings of having command line only.*◀

5.2 Reproducing and Navigating Program States

The Timelapse developer tool is designed around two activities: capturing user-demonstrated program behaviors, and quickly and repeatedly reaching program states within a recording when localizing and understanding a fault. The novel features we describe in this section support these activities by making it simple to record program behavior and by providing visualizations and affordances that quicken the process of finding and seeking to relevant program states without manually reproducing behavior.

To better understand the utility of replay capabilities during debugging, we first conducted a small pilot study with a prototype record/replay interface. Using contextual inquiry, we found that developers primarily used the prototype to isolate buggy output, and to quickly reach specific states when working backwards from buggy output towards its root cause. Towards these ends, we saw several common use cases: “play and watch”; isolating output using random-access seeking; stepping through execution in single-input increments, and reading low-level input details or logged output.

This section introduces the novel features of the Timelapse developer tool by showing how a fictional developer named Steph might use Timelapse’s features while debugging.

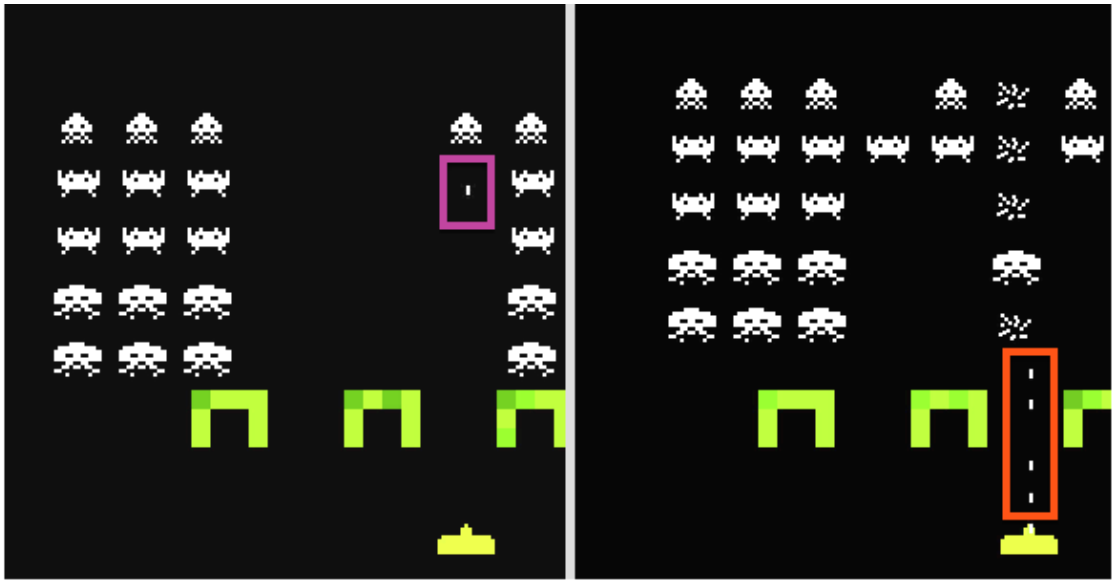

Figure 5.1: Screenshots of normal and buggy Space Invader game mechanics. Only one bullet should be in play at a time (shown on left). Due to misuse of library code, each bullet fires two asynchronous `bulletDied` events instead of one event when it is destroyed. The double dispatch sometimes enables multiple player bullets being in play at once (shown on right). This happens when two bullets are created: one between two `bulletDied` events, and the other after both events.

5.2.1 Debugging Scenario: (Buggy) Space Invaders

Steph, a new hire at a game company, has been asked to fix a bug in a web application version of the Space Invaders video game[1]. In this game, the player moves a defending ship and shoots advancing aliens. The game's implementation is representative of modern object-oriented interactive web programs: it uses timer callbacks, event-driven programming, and helper libraries. The game contains a defect that allows multiple player bullets to be in flight at a time; there is only supposed to be one player bullet at a time (Figure 5.1).

---

[1] `http://matthaynes.net/playground/javascript/glow/spaceinvaders/`

5.2.2 Reproducing Program Behavior

Steph is unfamiliar with the Space Invaders implementation, so her first step towards understanding and fixing the multiple-bullet defect is to figure out how to reliably reproduce it. This is difficult because the failure only occurs in specific game states and is influenced by execution conditions outside of her control, such as random numbers, the current time, or asynchronous network requests.

With Timelapse, Steph begins capturing program behaviors with a single click (Figure 5.2-6), plays the game until she reproduces the failure, and then finishes the recording. Recordings created by Timelapse are compact, self-contained, and serializable, so Steph can attach her recording to a bug report or share it via email.

To reproduce the defect with traditional tools, Steph would have to multitask between synthesizing reproduction steps, playing the video game, and reflecting on whether the reproduction steps are correct. Once Steph finds reliable reproduction steps, she could then use breakpoints to further understand the defect. But, breakpoints might themselves affect timing, making the defect harder to trigger or requiring modified reproduction steps.

5.2.3 Navigating to Specific Program States

To focus her attention on code relevant to the failure, Steph needs to know which specific user input—and thus which event handler invocations—caused the second bullet to appear.

Steph uses Timelapse's visualization and navigation tools (Figure 5.2) to locate and seek the recording to an instance of the multiple bullets failure. First, Steph limits the zoom interval to when she actually fired bullets, and then filters out non-keystroke inputs. She replays single keystrokes with a keyboard shortcut until a second bullet is added to the game board. Then, she seeks execution backward by one keystroke. At this point, she is confident that the code which created the second bullet ran in response to the current keystroke.

Without Timelapse, it would not be possible for Steph to isolate the failure to a specific keystroke and then work backwards from the keystroke. Instead, she would have to insert log-

ging statements, repeatedly reproduce the failure to generate logging output, and scrutinize logged values for clues leading towards the root cause.

5.2.4 Navigation Aid: Debugger Bookmarks

Having tracked down the second bullet to a specific user input, Steph now needs to investigate what code ran, and why.

With Timelapse, Steph sets several *debugger bookmarks* (Figure 5.3-6) at positions in the recording that she wants to quickly revert back to, such as the keystroke that caused the second bullet to appear or an important program point reached via the debugger. Debugger bookmarks support the concept of temporal focus points [51, 112], which are useful when a developer wants to relate information [56]—such as the program's state at a breakpoint hit—that is only available at certain points of execution. Timelapse restores a debugger bookmark by seeking to the preceding input, setting and continuing to the preceding breakpoint, and finally simulating the user's sequence of debugger commands (step forward/into/out).

With traditional tools, Steph must explore an execution with debugger commands such as "step into", "step over", and "step out". This is frustrating because these commands are irreversible, and Steph would have to manually reproduce the failure multiple times to compare multiple instants or the effects of different commands.

5.2.5 Navigation Aid: Breakpoint Radar

Once Steph finds the code that creates bullets, she needs to understand why some keystrokes fire bullets and others do not.

With Timelapse, Steph first adjusts the zoom interval to include keystrokes that did and did not trigger the failure. Then, she sets a breakpoint inside the `Bullet.create()` method and records and visualizes when it is actually hit during the execution using the *breakpoint radar* feature (Figure 5.3-3a). Breakpoint radar automates the process of replaying the execution, pausing and resuming at each hit, and visualizing when the debugger paused. Steph can easily see which keystrokes

created bullets and which did not.

With traditional tools, Steph would need to repeatedly set and unset breakpoints in order to determine which keystrokes did or *did not* create bullets. To populate the breakpoint radar timeline, Steph would have to manually hit and continue from dozens or hundreds of breakpoints, and collect and visualize breakpoint hits herself. For this particular defect, breakpoints interfere with the timing of the bullet's frame-based animations, so it would be nearly impossible for Steph to pause execution when two bullets are in flight.

5.2.6 Interacting with Other Debugging Tools

Once Steph localizes the part of the program responsible for the multiple bullets, she still needs to isolate and fix the root cause. To do so, Steph uses debugging strategies that do not require Timelapse, but nonetheless benefit from it. Timelapse is designed to be used alongside other debugging tools such as breakpoints[2], logging, and element inspectors; its interface can be juxtaposed (Figure 5.3) with other tools.

Through code inspection, Steph observes that the creation of a bullet is guarded by a flag indicating whether any bullets are already on the game board. The flag is set inside the `Bullet.create()` method and cleared inside the `Bullet.die()` method. To test her intuition about the code's behavior, she inserts logging code and captures a new execution to see if the method calls are balanced. The logging output in Figure 5.4 is synchronized with the replay position: as Steph seeks execution forward or backward, logging output up to the current instant is displayed. Logging output is cleared when a fresh execution begins (i.e., Timelapse seeks backwards) and then populated as the program executes normally.

Steph has discovered that the multiple-bullet defect is caused by the `bulletDied` event being fired twice, allowing a second replacement bullet to be created if the bullet "fire" key is pressed between the two event dispatches. In other words, the failure is triggered by firing a bullet while another bullet is being destroyed (by collision or leaving the game board).

---

[2]To prevent breakpoints from interfering with tool use cases, Timelapse tweaks breakpoints in several ways: breakpoints are disabled when recording or seeking, and enabled during real-time playback.

With basic record/replay functionality, affordances for navigating the stream of recorded inputs, and the ability to easily reach program states by jumping directly to breakpoint hits, Timelapse both eliminates the need for Steph to repeatedly reproduce the Space Invaders failure and frees her to focus on understanding the program's logic. Our user evaluation explores these benefits further.

Figure 5.2: The Timelapse tool interface presents multiple linked views of recorded program inputs. Above, the timelines drawer (1) is juxtaposed with a detailed view of program inputs (2). The recording overview (3) shows inputs over time with a stacked line graph, colored by category. The overview's selected region is displayed in greater detail in the heatmap view (4). Circle sizes indicate input density per category. In each view, the red cursor (5) indicates the current replay position and can be dragged. Buttons allow for recording (6), 1× speed replay (7), and breakpoint scanning (8). Details for the selected circle (9a) are shown in a side panel (9b).

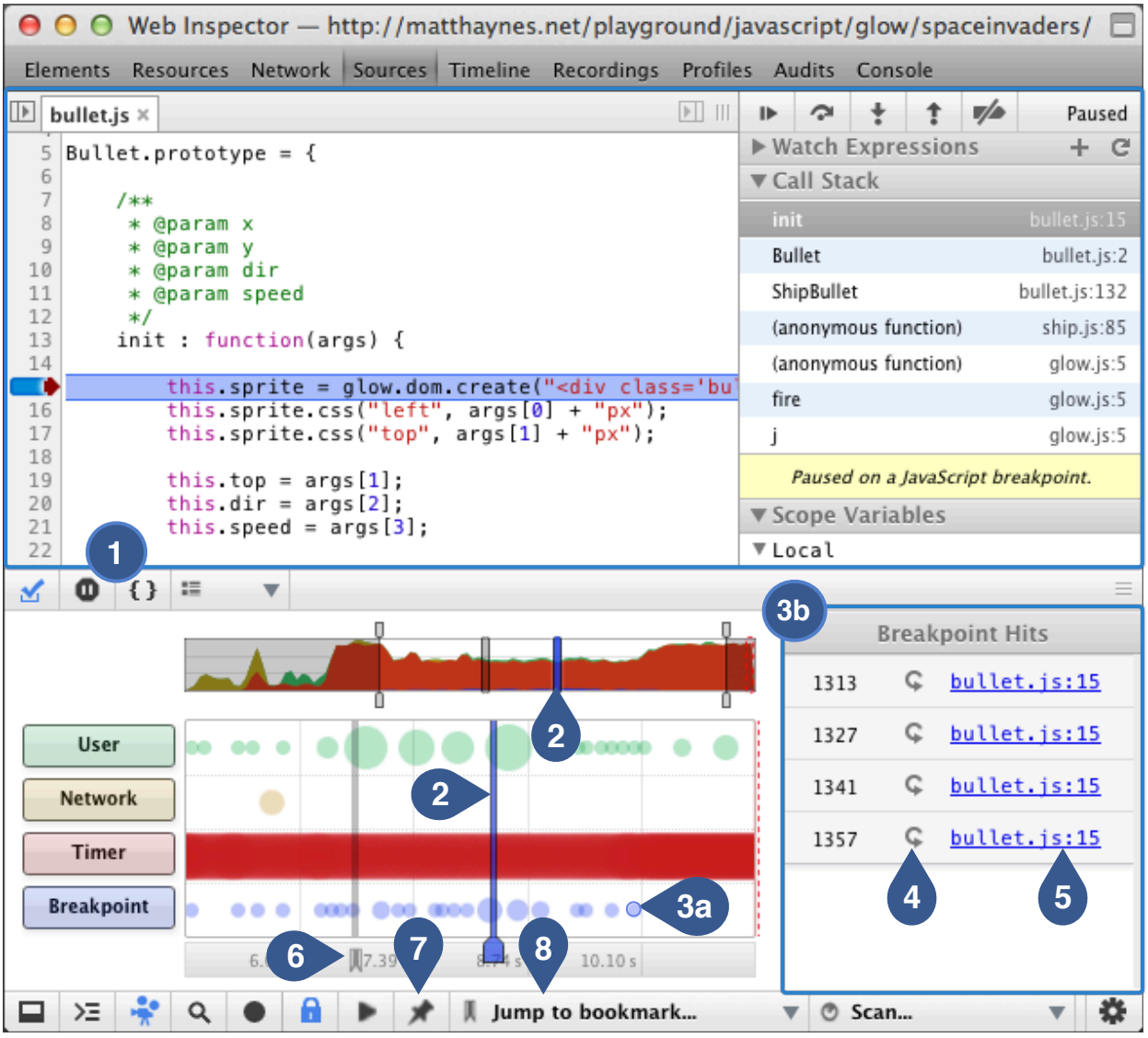

Figure 5.3: Timelapse's visualization of debugger status and breakpoint history, juxtaposed with the existing Sources panel and debugger (1). A blue cursor (2) indicates that replay execution is paused at a breakpoint, instead of between user inputs (as shown in Figure 5.2). Blue circles mark the location of known breakpoint hits, and are added or removed automatically as breakpoints change. A side panel (3b) shows the selected (3a) circle's breakpoints. Shortcuts allow for jumping to a specific breakpoint hit (4) or source location (5). Debugger bookmarks (6) are set with a button (7) and replayed to by clicking (6) or by using a drop-down menu (8).

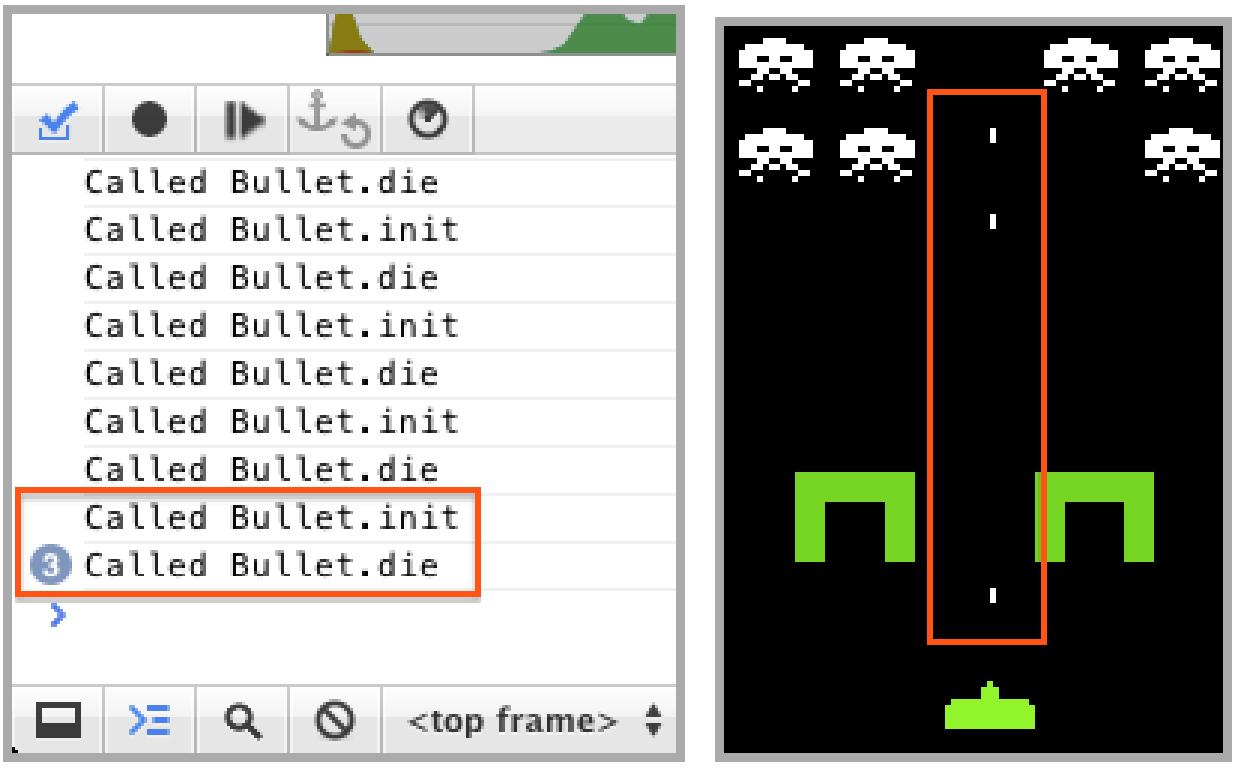

Figure 5.4: Screenshots of the logging output window and the defect's manifestation in the game. Steph added logging statements to the `die` and `init` methods. At left, the logging output shows the order of method entries, with the blue circle summarizing 3 identical logging outputs. According to the last 4 logging statements (outlined in red), calls to `die` and `init` are unbalanced. At right, three in-flight bullets correspond to the three calls to `Bullet.init`.

Chapter 6

HOW DO DEVELOPERS USE TIMELAPSE?

Prior work [33, 80, 109] asserts the usefulness of deterministic record and replay for debugging. In this section, we present a formative user study that investigates when, how, and for whom record/replay tools and specifically Timelapse are beneficial. Our specific research questions were:
RQ 1 How does Timelapse affect the way that developers reproduce behavior?
RQ 2 How do successful and unsuccessful developers use Timelapse differently?

6.1 Study Design

We recruited 14 web developers, 2 of which we used in pilot studies to refine the study design. Each participant performed two tasks. We used a within-subjects design to see how Timelapse changed the behavior of individual developers. For one task, participants could use the standard debugging tools included with the Safari web browser. For the other task, they could also use Timelapse. To mitigate learning effects, we randomized the ordering of the two tasks, and randomized task in which they were allowed to use Timelapse.

The goal of this study was to explore the variation in how developers used Timelapse, so the tasks needed to be challenging enough to expose a range of task success. To balance realism with replicability, we chose two tasks of medium difficulty, each with several intermediate milestones. Based on our results, our small exploratory study was still sufficiently large to capture the variability in debugging and programming skill among web developers.

6.2 Participants

We recruited 14 web developers in the Seattle area. Each participant had recently worked on a substantial interactive website or web application. One half of the participants were developers,

designers, or testers. The other half were researchers who wrote web applications in the course of their research. We did not control for experience with the jQuery or Glow libraries used by the programs being debugged.

6.3 Programs and Tasks

6.3.1 Space Invaders

One program was the Space Invaders game from our earlier example scenario. The program consists of 625 SLOC in 6 files (excluding library code) and uses the Glow JavaScript library[1]. We chose this program for two reasons: its extensive use of timers makes it a heavy record/replay workload, and its event-oriented implementation is representative of object-oriented model-view programs, the dominant paradigm for large, interactive web applications.

We asked participants to fix two Space Invaders defects. The first was an API mismatch that occurred when we upgraded the Glow library to a recent version while preparing the program for use in our study. In prior versions, a sprite's coordinates were represented with `x` and `y` properties; in recent versions, coordinates are instead represented with `left` and `top` properties, respectively. After upgrading, the game's hit detection code ceases to work because it references the obsolete property names. The second defect was described in the motivating example and was masked by the first defect.

6.3.2 Colorpicker

The other program was Colorpicker[2], an interactive widget for selecting colors in the RGB and HSV colorspaces (see in Figure 6.1). The program consists of about 500 LOC (excluding library and example code). The widget supports color selection via RGB (red, green, blue) or HSV (hue, saturation, brightness) component values or through several widgets that visualize dimensions of the HSV colorspace.

We chose this program because it makes extensive use of the popular jQuery library, which—by virtue of being highly layered, abstracted, and optimized—makes reasoning about and following

[1] `http://www.bbc.co.uk/glow/`

[2] `http://www.eyecon.ro/colorpicker/`

the code significantly more laborious.

The Colorpicker task was to create a regression test for a real, unreported defect in the Colorpicker widget. The defect manifests when selecting a color by adjusting an RGB component value, as shown in Figure 6.1. If the user drags the G component (left panel, orange), the R component spontaneously changes (right panel, red). The R component should not change when adjusting the G component. The bug is caused by unnecessary rounding in the algorithm that converts values between RGB and HSV color spaces. Since the color picker uses the HSV representation internally, repeated conversions between RGB and HSV can expose numerical instability during certain patterns of interaction.

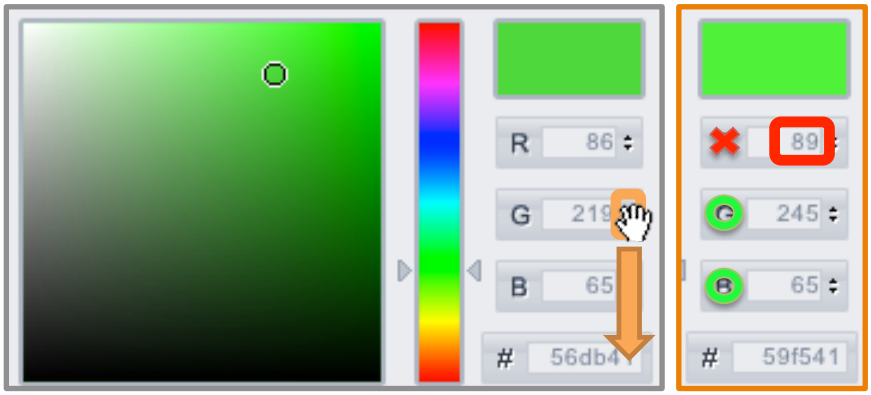

Figure 6.1: The Colorpicker widget.

We claim that both of these faults are representative of many bugs in interactive programs. Often there is nothing wrong with the user interface or event handling code *per se*, but faults that are buried deep within the application logic are only uncovered by user input or manifest as visual output. The Space Invaders faults lie in incorrect uses of library APIs, but manifest as broken gameplay mechanics. Similarly, the Colorpicker fault exists in a core numerical routine, but is only manifested ephemerally in response to mouse move events.

6.4 Procedure

Participants performed the study alone in a computer lab. Participants were first informed of the general purpose and structure of the study, but not of our research questions to avoid observer and subject expectancy effects. Immediately prior to the task where Timelapse was available, participants spent 30 minutes reading a Timelapse tutorial and performing exercises on a demo

program. In order to proceed, participants were required to demonstrate mastery of recording, replaying, zooming, seeking, and using breakpoint radar and debugger bookmarks. Participants could refer back to the tutorial during subsequent tasks.

Each task was described in the form of a bug report that included a brief description of the bug and steps to reproduce the fault. At the start of each task, the participant was instructed to read the entire bug report and then reproduce the fault. Each task was considered complete when the participant demonstrated their correct solution. Participants had up to 45 minutes to complete each task. They were not asked to think aloud[3].

.

We stopped participants when they had demonstrated successful completion to us or exceeded the time limit.

After both task periods were over, we interviewed participants for 10 minutes about how they used the tool during the tutorial and tool-equipped task and how they might have used the tool on the other task. We also asked about their prior experience in bug reproduction, debugging, and testing activities. Participants who completed the study were compensated for their time.

6.5 Data Collection and Analysis

We captured a screen and audio recording of each participant's session, and gathered timing and occurrence data by reviewing the video recordings after the study concluded.

Our tasks were both realistic and difficult so as to draw out variations in debugging skill and avoid imposing a performance ceiling. We measured task success via completion of several intermediate task steps or critical events. For the Space Invaders task, the steps were: successful fault reproduction, identifying the API mismatch, fixing the API bug, reproducing the rate-of-fire defect, written root cause, and fixing the rate-of-fire defect. For the Colorpicker task, the steps were: successful fault reproduction, written root cause, correct test form, identifying a buggy input, and verifying the test.

[3]In an earlier formative study, we solicited design feedback by using a think aloud protocol. We did not do so in the present study to avoid biasing participants' work style.

We measured the time on task as the duration from the start of the initial reproduction attempt until the task was completed or until the participant ran out of time. We recorded the count and duration of all reproduction activities and whether the activity was mediated by Timelapse (automatic reproduction) or not (manual reproduction). Reproduction times only included time in which participants' attention was engaged on reproduction, which we determined by observing changes in window focus, mouse positioning, and interface modality.

6.6 Results

Below, we summarize our findings of how Timelapse affects developers' reproduction behavior (RQ1) and how this interacts with debugging expertise (RQ2).

Timelapse did not reduce time spent reproducing behaviors. There was no significant difference in the percentage of time spent reproducing behaviors across conditions and tasks. Though Timelapse makes reproduction of behaviors simpler, it does not follow that this fact will reduce overall time spent on reproduction. We observed the opposite: because reproduction with Timelapse was so easy, participants seemed more willing to reproduce behaviors repeatedly. A possible confound is that behaviors in our tasks were fairly easy to reproduce, so Timelapse only made reproduction less tedious, not less challenging. We had hoped to test whether Timelapse is more useful for fixing more challenging bugs, but were forced to reduce task difficulty so that we could retain a within-subjects study design while minimizing participants' time commitment.

8–25% of time was spent reproducing behavior. Even when provided detailed and correct reproduction steps, developers in both conditions spent up to 25% (and typically 10–15%) of their time reproducing behaviors. Participants in all tasks and conditions reproduced behavior many times (median of 22 instances) over small periods. This suggests that developers frequently digress from investigative activities to reproduce program behavior. These measures are unlikely to be ecologically valid because most participants did not complete all tasks, and time spent on reproduction activities outside of the scope of our study tasks (i.e., during bug reporting, triage, and testing) is not included.

Expert developers incorporated replay capabilities. High-performing participants—those

who successfully completed the most task steps—seemed to better integrate Timelapse's capabilities into their debugging workflows. Corroborating the results of previous studies [92, 105], we observed that successful developers debugged methodically, formed and tested debugging hypotheses using a bisection strategy, and revised their assessment of the root cause as their understanding of the defect grew. They quickly learned how to use Timelapse to facilitate these activities. They used Timelapse to accelerate familiar tasks, rather than redesigning their workflow around record/replay capabilities. In the Colorpicker task, participant 1 used Timelapse to step through each change to the widget's appearance to isolate a buggy RGB value. Participants in the control condition appeared to spend much more time finding this buggy value, since they had to isolate a small change by interacting carefully with the widget. Participant 11 used Timelapse to compare program state before and after each call in the `mousemove` event handler, and then used Timelapse to move back and forth in time when bisecting the specific calls that caused the widget's RGB values to update incorrectly. Participants in the control condition appeared to achieve the same strategy more slowly by interleaving changes to breakpoints and manual reproduction.

Timelapse distracted less-successful developers. Those who only achieved partial or limited success had trouble integrating Timelapse into their workflow. We partially attribute this to differences in participants' prior debugging experiences and strategies. The less successful participants used ad-hoc, opportunistic debugging strategies; overlooked important source code or runtime state; and were led astray by unverified assumptions. Consequently, even when these developers used Timelapse, they did not use it to a productive end.

6.7 Discussion

todo ►*expand*◄ In our study, developers used Timelapse to automatically reproduce program behavior during debugging tasks, but this capability alone did not significantly affect task times, task success, or time spent reproducing behaviors. For developers who employed systematic debugging strategies, Timelapse was useful for quickly reproducing behaviors and navigating to important program states. Timelapse distracted developers who used ad-hoc or opportunistic strategies, or who were unfamiliar with standard debugging tools. Timelapse was used to accelerate the repro-

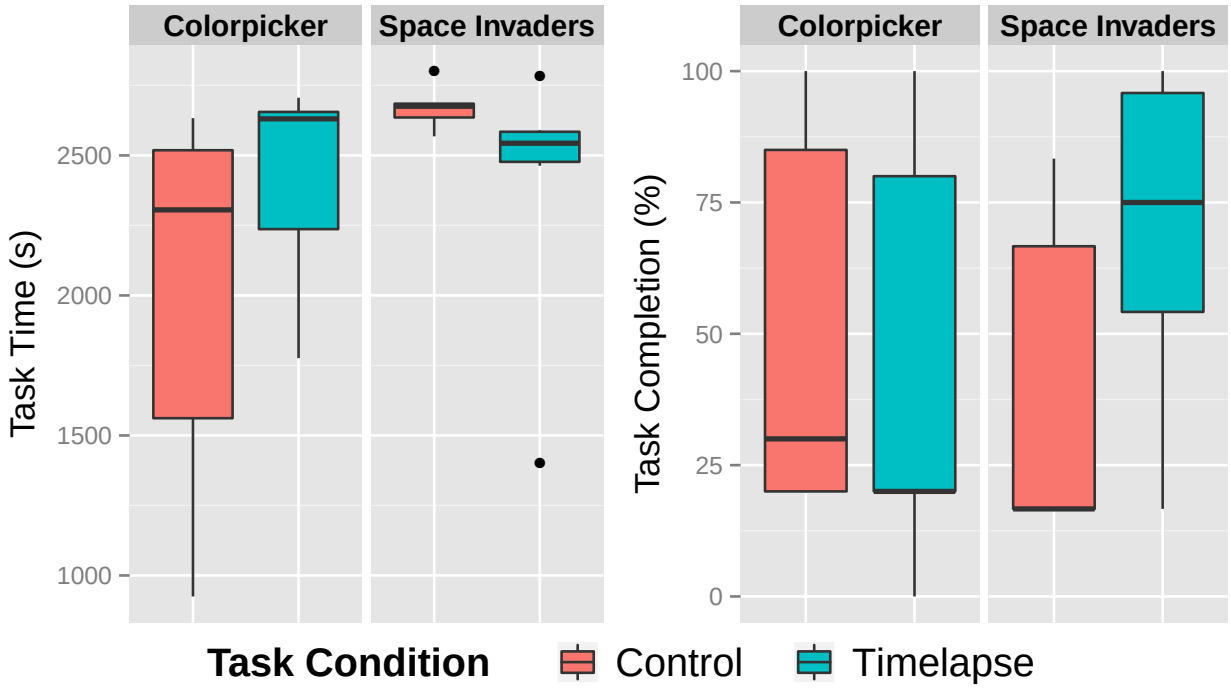

Figure 6.2: A summary of task time and success per condition and task. Box plots show outliers (points), median (thick bar), and 1st and 3rd quartiles (bottom and top of box, respectively). There was no statistically significant difference in performance between participants who used standard tools and those who had access to Timelapse.

duction steps of existing strategies, but did not seem to affect strategy selection during our short study. As with any new tool, it appears that some degree of training and experience is necessary to fully exploit the tool's benefits. In our small study, the availability of Timelapse had no statistically significant effects on participants' speed or success in completing tasks. Figure 6.2 shows task success and task time per task and condition. In future work, we plan to study how long-term use of Timelapse during daily development affects debugging strategies. We also plan to investigate how recordings can improve bug reporting practices and communication between bug reporters and bug fixers.

Chapter 7

EXPLAINING VISUAL CHANGES IN WEB INTERFACES

7.1 Stuff from UIST Submission

7.2 Introduction

Web developers increasingly look to existing web site designs for inspiration [34], to learn about new practices or APIs [13], and to copy and adapt interactive behaviors for their own purposes [116]. 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 [59]. 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 [91], 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 [3, 15, 80], 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 [15, 67]. In other programming environments, research prototypes with this functionality all incur run-time overheads that limit their use to post-mortem debugging [47, 55, 88].

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. 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.

7.3 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 7.1) 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/`

Figure 7.1: 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.

7.3.1 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 7.2.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 7.2.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 7.2.b). For each selected screenshot, Scry shows the small subset of the page's DOM (Figure 7.2.b) and CSS (Figure 7.2.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 7.2.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 7.2.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 7.2.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.

7.4 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 [108]. Scry's interface simplifies this work by identifying and supporting three distinct activities through dedicated interfaces: (1) the 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.

7.4.1 Design Rationale

We designed Scry to directly address the information overload a developer encounters when performing feature location tasks [108]. 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 7.3). 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 7.3).

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 7.4). 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 from which users can plug in their own search and navigation strategies [67] 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

[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 7.5), 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.

7.4.2 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 7.2.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 7.3.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.

7.4.3 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 7.3.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 7.3). 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 7.3.c), Scry shows the element's matched CSS styles (Figure 7.3.d).

7.4.4 Comparing Internal States

Using Scry's comparison interface (Figure 7.4), 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 7.4) 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.

7.4.5 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

[3]Scry does not directly explain causal relationships between inputs and outputs in the style of Whyline [55]. 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 7.4.a). Then, Scry changes views to show the difference alongside a list of mutation operations (Figure 7.5.b) that caused the difference. Each operation includes a JavaScript stack trace that shows the calling context for the mutation operation (Figure 7.5.c). Using this link, a developer can find pieces of code related to a single visually-significant difference.

7.5 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.

7.5.1 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 [91] 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.

7.5.2 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 style properties originate from one of four sources: declarative *style rules*, explicit *inline styles*, CSS animations, and inherited properties. In

[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.

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 [15] 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.

7.5.3 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

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

strategy identifies many small, localized changes (Table 7.2) that are straightforward to explain in terms of low-level mutation operations (Table 7.1). 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.

An alternative strategy for comparing subtrees is to globally summarize changes using tree matching algorithms [59] or tree edit distance algorithms [9]. 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,

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 7.2: 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.

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.

7.5.4 Explaining State Differences

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 7.1. 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 7.1) 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 7.2), 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].

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.

[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.

7.5.5 Prototype Implementation

Scry is implemented as a set of modifications to the WebKit browser engine [134] 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.

7.6 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.

7.6.1 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 in-

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

teracted 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.

7.6.2 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 [37] expose an immediate-mode API called the *virtual DOM*. Client JavaScript implements `draw()` methods that fully recreate a widget's DOM

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

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 similar nodes, but could be extended to fall back to using more relaxed similarity-based metrics [59] instead of strict identity when re-finding DOM elements.

7.6.3 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

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

"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 [111]. This would require a very different technical approach, since Scry instruments native browser APIs rather than JavaScript code.

7.7 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* [32] and *program comprehension* [24]—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.

7.7.1 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 [19], 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 [128], 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.

7.7.2 Explaining How Interactive Behaviors Work

Scry follows a long line of methods and tools [113] that aim to help a developer comprehend specific program features or behaviors. Recent work for user interfaces has focused on inferring

behavioral models [1, 79, 81], logging and visualizing user inputs and runtime events [15, 91], and using program analysis to produce causal explanations of behaviors [55, 111]. Scry shares the same reverse-engineering goals as FireCrystal [91], 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 [55]. 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 [136] 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 [10, 139] 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.

7.8 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 [10]. Concretely, Scry could delete

individual style properties from a snapshot if the resulting visual output does not differ [139]. Even with a minimal set of inputs, a more involved "white-box" approach to explaining layout [85] 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 [54, 55, 85] 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 [111] and event modeling [1] to extend its explanations to show an uninterrupted causal chain [55] 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 [15] to collect snapshots and trace data from earlier instants of the execution. Prior work has demonstrated the feasibility of such an "offline dynamic analysis" [20, 21, 111], but none has integrated this technique into a user interface or web browser.

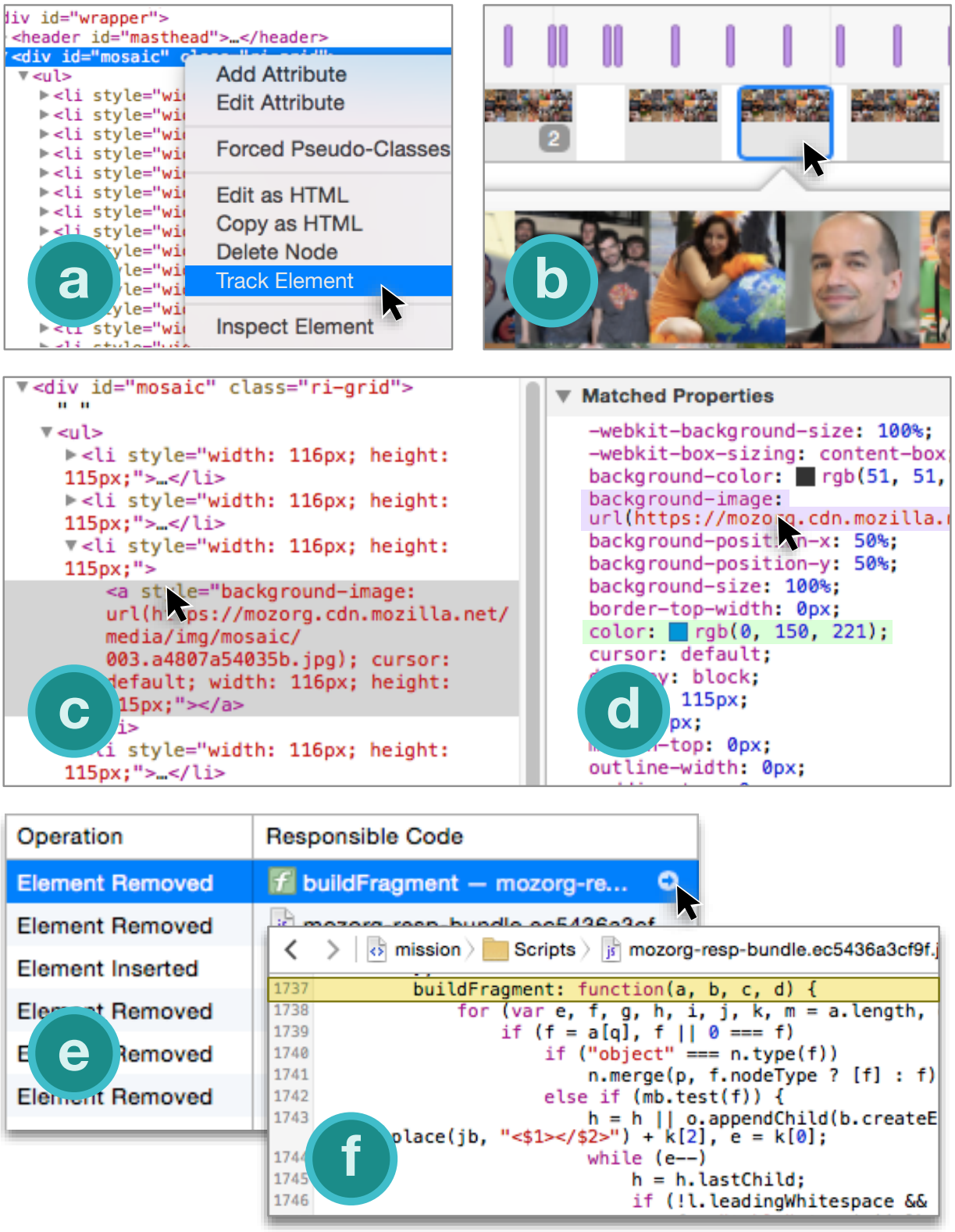

Figure 7.2: Susan first uses the Web Inspector to go from the mosaic's visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual

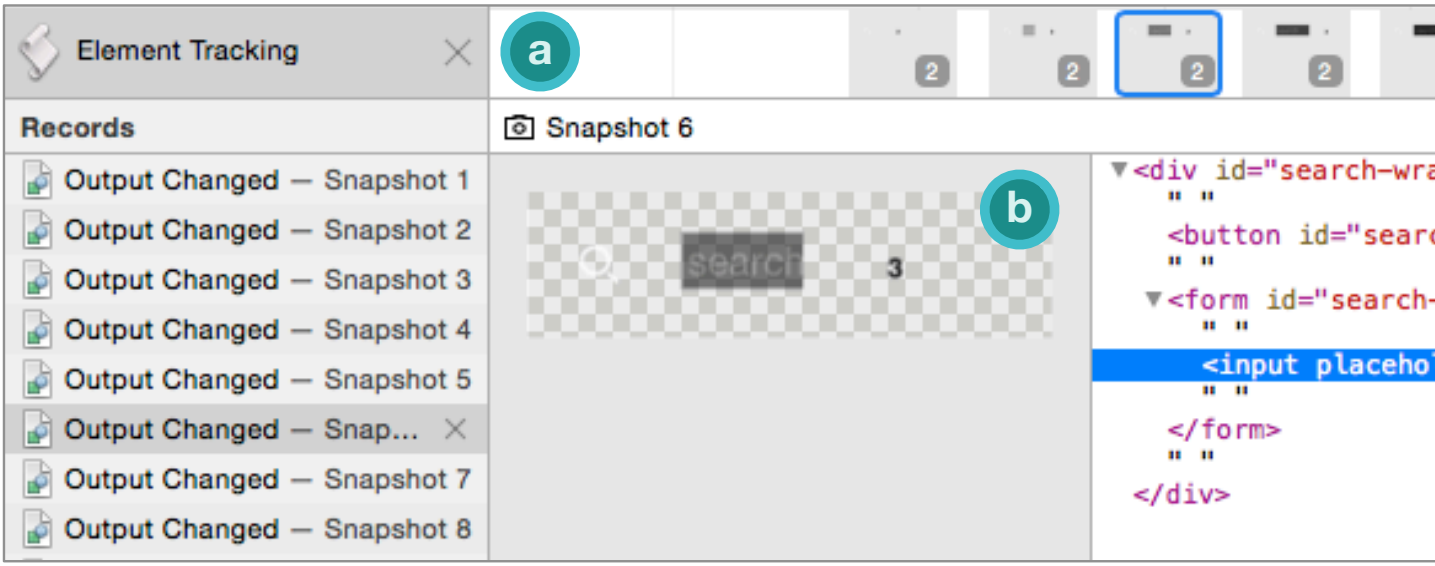

Figure 7.3: Scry's snapshot interface shows multiple screenshots in a timeline view (a). When the developer selects an output example from the timeline, Scry shows three views for it: (b) the element's visual appearance, (c) its corresponding DOM tree, and (d) computed style properties for the selected DOM node.

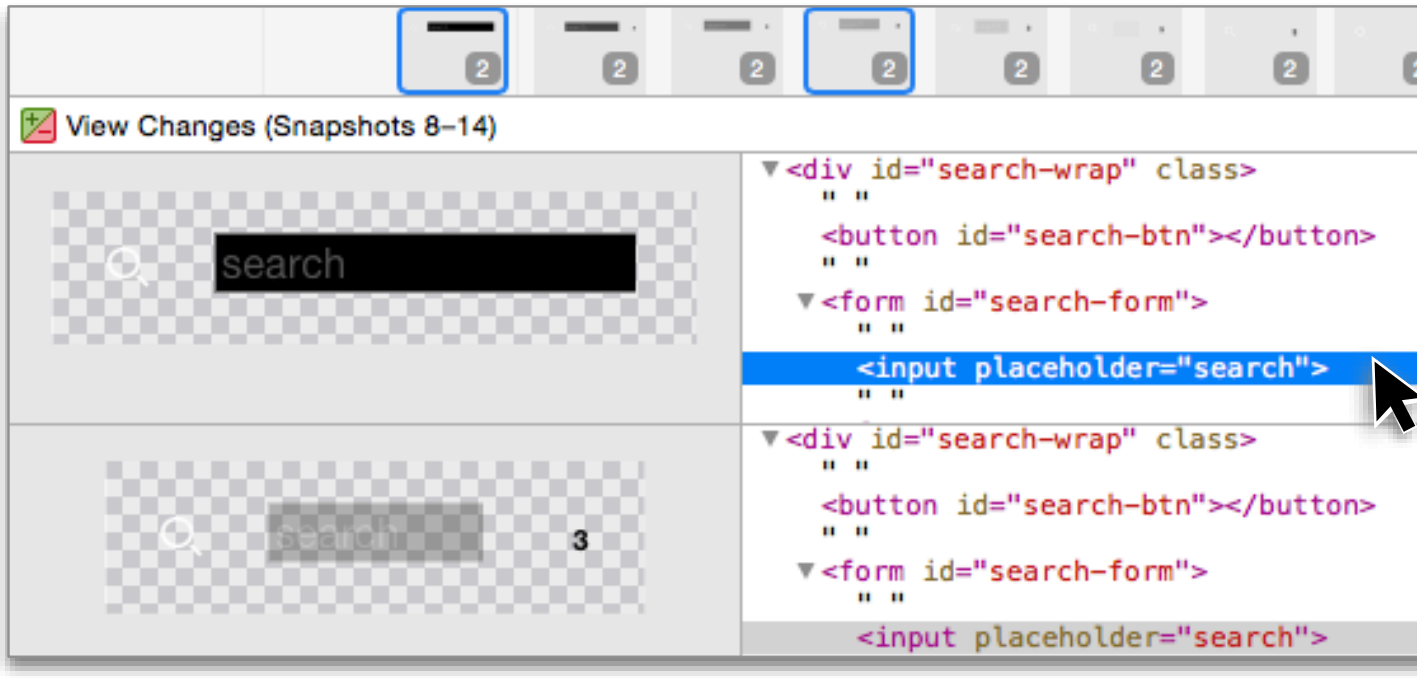

Figure 7.4: The snapshot comparison interface. Two screenshots are selected in the timeline, and their corresponding CSS styles and DOM trees appear below. Differences are highlighted using inline annotations; here, the `opacity` style property has changed. Additions, removals, and changes are highlighted in green, red, and purple, respectively.

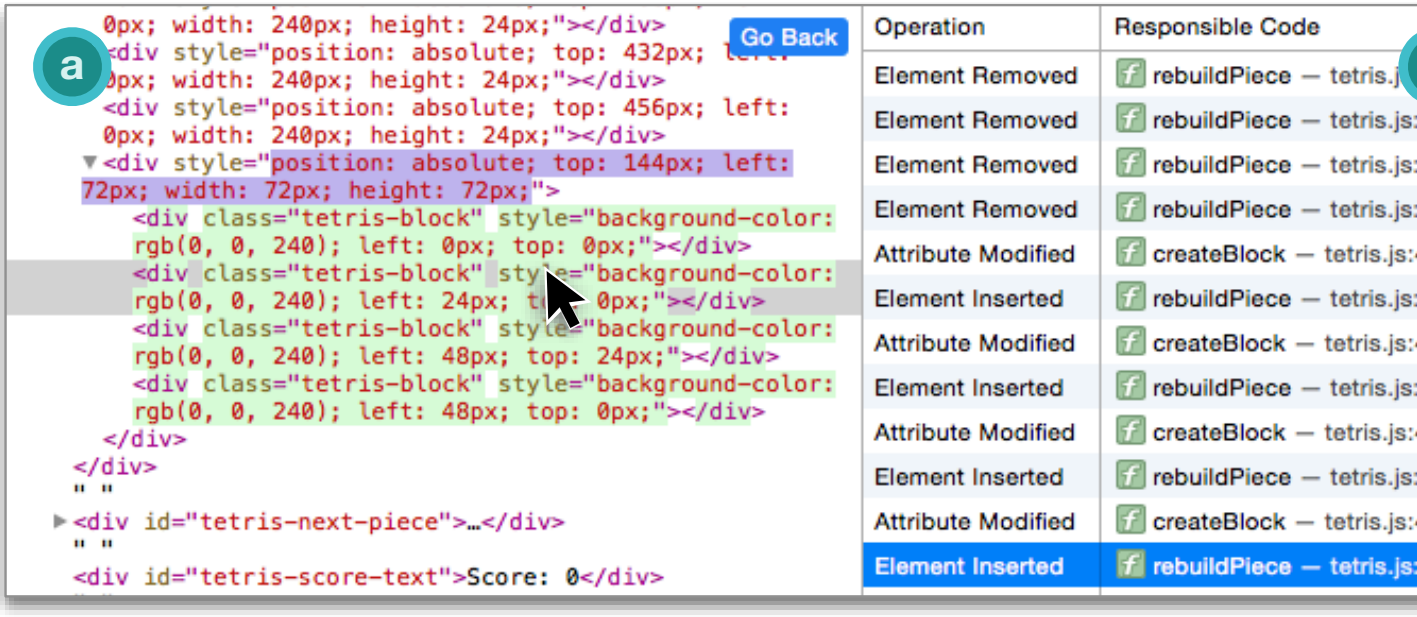

Figure 7.5: Explaining a DOM tree difference in a Tetris game that modifies, removes, and rebuilds DOM elements. The left pane (a) shows a single difference being investigated (the second added `<div>`). The center pane (b) shows a list of mutation operations that caused the change. When an operation is selected, Scry shows a source code preview (c) and call stack (not shown) for the operation.

Chapter 8

FUTURE WORK

There are several contributions which have been partially developed, but whose dissemination is not planned within my graduation timeline.

Dolos/Timelapse journal article. Since the UIST paper, additional infrastructure work has been ongoing to support the development of Timelapse, Scry, Overscan, and upstreaming Dolos into WebKit [14]. The new contributions, industrial experience, and unpublished details are significant enough for an article. It would be easy to adapt dissertation chapters into a journal submission.

Overscan conference paper. Depending on the significance of Overscan's contributions, we could write a second paper that focuses on instrumentation. A paper would include the same account of Overscan's design and implementation as the dissertation, but require a more thorough evaluation, such as implementing dynamic analyses proposed in related work and measuring performance.

Upstreaming to WebKit. Work towards upstreaming Dolos and Timelapse into the WebKit codebase will not take priority over research tasks described here, except to reduce bitrot and painful merges. There are no plans to upstream Scry/Overscan.

This dissertation leaves as future work many applications of our deterministic replay infrastructure. For example, we would like to explore its use in reproducing field failures [50]; to investigate variability in software [36]; to distill performance issues into regression tests; to explore controlled divergence [60, 129]; to enable distributed program understanding; to automatically isolate, minimize [68], and fix failures [89**?**], and other topics.

Chapter 9

CONCLUSION

This dissertation develops several enabling technologies for retroactively understanding dynamic behavior. First, we propose Dolos, a novel deterministic replay infrastructure that targets the highly dynamic, visual, and interactive domain of web applications. Dolos is the first such infrastructure to adapt precise, low-overhead virtual machine replay techniques to modern browser runtimes. Second, we propose Timelapse, a suite of affordances and algorithms for discovering and navigating to important program states within a captured execution. Timelapse introduces the concept of *probe points* and *time-indexed events* as affordances for navigating through a recording via its output. Finally, we propose Scry and Overscan, a developer tool and supporting instrumentation framework for iteratively and retroactively investigating dynamic control flow. Scry shows how a developer can answer high-level questions about control flow via novel interactive visualization that gathers runtime data on demand. Overscan is the first dynamic instrumentation framework for JavaScript that is explicitly designed to exploit deterministic execution and contextual information.

Through the contributions of this dissertation, we hope to demonstrate the utility of replay infrastructures for program comprehension so that their value is more widely appreciated by platform and tool developers.

BIBLIOGRAPHY

[1] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. 2014.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. 2009.

[3] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. 2011.

[4] D. Ansaloni, W. Binder, C. Bockisch, E. Bodden, K. Hatun, L. Marek, Z. Qi, A. Sarimbekov, A. Sewe, P. Tůma, and Y. Zheng. Challenges for refinement and composition of instrumentations: Position paper. 2012.

[5] K. Arya, T. Denniston, A.-M. Visan, and G. Cooperman. Semi-automated debugging via binary search through a process lifetime. 2013.

[6] BCEL Contributors. Apache Commons BCEL, 2014. `http://commons.apache.org/proper/commons-bcel/`.

[7] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. 2006.

[8] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. 2014.

[9] P. Bille. A survey on tree edit distance and related problems. *Theoretical Comput. Sci.*, 337, June 2005.

[10] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: language-independent program slicing. 2014.

[11] B. Boothe. Efficient algorithms for bidirectional debugging. 2000.

[12] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. Laviola Jr. Code Bubbles: rethinking the user interface paradigm of integrated development environments. 2010.

[13] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. 2009.

[14] B. Burg. Announcement: web replay support, 2014. `https://lists.webkit.org/pipermail/webkit-dev/2014-January/026062.html`.

[15] B. Burg, R. J. Bailey, A. J. Ko, and M. D. Ernst. Interactive record and replay for web application debugging. 2013.

[16] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. 2004.

[17] T. Cardenas, M. Bastea-Forte, A. Ricciardi, B. Hartmann, and S. R. Klemmer. Testing physical computing prototypes through time-shifted and simulated input traces. 2008.

[18] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: How and why developers use drawings. 2007.

[19] P. K. Chilana, A. J. Ko, and J. O. Wobbrock. LemonAid: selection-based crowdsourced contextual help for web applications. 2012.

[20] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. 2008.

[21] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with Crosscut. 2010.

[22] J. Clouse and A. Orso. Camouflage: Automated anonymization of field data. 2011.

[23] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. de Bosschere. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[24] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. 36, Sept./Oct. 2009.

[25] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. 44:97–102, Jan. 2010.

[26] R. DeLine and K. Rowan. Code Canvas: Zooming towards better development environments. 2010.

[27] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: industrial experience with the Code Bubbles paradigm. 2012.

[28] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. 6(9):275–295, 2007.

[29] F. Détienne. *Software Design - Cognitive Aspects*. Springer-Verlang, 2001.

[30] P. Dickinson. Instant replay: Building a game engine with reproducible behavior. In *Gamasutra*, July 2001. `http://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php`.

[31] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *In Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996.

[32] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25:53–95, 2013.

[33] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. 36(SI):211–224, Dec. 2002.

[34] C. Eckert and M. Stacey. Sources of inspiration: a language of design. *Design Studies*, 21 (5):523–538, 2000.

[35] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. SeeSoft—a tool for visualizing line oriented software metrics. 18:957–968, Nov. 1992.

[36] EUSES Consortium. Variations to support exploratory programming, 2014. `http://www.exploratoryprogramming.org/`.

[37] Facebook. React: a JavaScript library for building user interfaces, 2015. `https://facebook.github.io/react/index.html`.

[38] GDB Contributors. Inferiors and programs, 2014. `http://sourceware.org/gdb/onlinedocs/gdb/Inferiors-and-Programs.html`.

[39] K. Glerum, K. Kinshuman, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and practice. 2009.

[40] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. 2005.

[41] T. Grossman, J. Matejka, and G. Fitzmaurice. Chronicle: capture, exploration, and playback of document workflow histories. 2010.

[42] P. J. Guo. Online Python Tutor: embeddable web-based program visualization for CS Education. 2013.

[43] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, W. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. 2008.

[44] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[45] J. Heer, J. D. Mackinlay, C. Stolte, and M. Agrawala. Graphical histories for visualization: supporting analysis, communication, and evaluation. 14(6):1189–1196, 2008.

[46] M. Hertzum and A. M. Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. 36:761–778, 2000.

[47] M. Hilgart. Step-through debugging of GLSL shaders, 2006.

[48] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: towards a new foundation for human-computer interaction research. 7:174–196, 2000.

[49] D. F. Jerding and J. T. Stasko. The information mural: a technique for displaying and navigating large information spaces. 1995.

[50] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. 2012.

[51] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. 2011.

[52] J. Kato, S. McDirmid, and X. Cao. DejaVu: integrated support for developing interactive camera-based programs. 2012.

[53] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. 2005.

[54] A. J. Ko and B. A. Myers. Designing the WhyLine: a debugging interface for asking questions about program behavior. 2004.

[55] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about Java program output. 20(2):1–36, Sept. 2010.

[56] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. 32(12):971–987, Dec. 2006.

[57] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. 2007.

[58] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. 2006.

[59] R. Kumar, J. O. Talton, S. Ahmad, and S. S. Klemmer. Bricolage: example-based retargeting for web design. 2011.

[60] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocesor operating systems. 2010.

[61] T. D. LaToza and B. A. Myers. Developers ask reachability questions. 2010.

[62] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. 2010.

[63] T. D. LaToza and B. A. Myers. Designing useful tools for developers. 2011.

[64] T. D. LaToza and B. A. Myers. Visualizing call graphs. 2011.

[65] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. 2006.

[66] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. 2008.

[67] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. 39(2):197–215, Feb. 2013.

[68] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Towards generating reducible replay logs. 2011.

[69] G. Lefebvre. *Composable and Reusable Whole-System Offline Dynamic Analysis*. PhD thesis, 2012.

[70] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. 2009.

[71] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. 2010.

[72] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: Automating and sharing how-to knowledge in the enterprise. 2008.

[73] B. Lewis. Debugging backwards in time. 2003.

[74] T. Lieber, J. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. 2014.

[75] L. Marek, A. Villazón, D. Zheng, Yudi anad Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. 2012.

[76] L. Marek, S. Kell, Y. Zheng, B. Lubomír, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: robust and comprehensive dynamic program analysis for the Java platform. 2013.

[77] S. McDirmid. Usable live programming. 2013.

[78] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. 2009.

[79] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. 6(1):1–30, 2012.

[80] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for JavaScript applications. 2010.

[81] M. Mirzaaghaei and A. Mesbah. DOM-based test adequacy criteria for web applications. 2014.

[82] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. 2011.

[83] Mozilla contributors. Mozilla Firefox web browser, 2014. http://www.mozilla.org/en-US/firefox/desktop/.

[84] Mozilla Contributors. rr: lightweight recording & deterministic debugging, 2014. https://rr-project.org/.

[85] B. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions is user interfaces. 2006.

[86] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. 2007.

[87] M. W. Newman, M. S. Ackerman, J. Kim, A. Prakash, Z. Hong, J. Mandel, and T. Dong. Bringing the field into the lab: Supporting capturing and RePlay of contextual data for design. 2010.

[88] R. O'Callahan. Efficient collection and storage of indexed program traces, 2006. http://www.ocallahan.org/Amber.pdf.

[89] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. 2014.

[90] J. Odvarko and Firebug contributors. Firebug: web development evolved, 2013. http://www.getfirebug.com/.

[91] S. Oney and B. Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. 2009.

[92] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? 2011.

[93] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through interactivity: online analysis of run-time behavior. 2010.

[94] M. Petre, A. F. Blackwell, and T. R. G. Green. Cognitive questions in software visualization. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization Programming as a Multi-Media Experience*, pages 453–480. MIT Press, Jan. 1998.

[95] D. Piorkowski, S. D. Fleming, I. Kwan, M. Burnett, C. Scaffidi, R. Bellamy, and J. Jordhal. The whats and hows of programmers' foraging diets. 2013.

[96] G. Pothier and E. Tanter. Extending omniscient debugging to support Aspect-oriented programming. 2008.

[97] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. 2011.

[98] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. 2007.

[99] Project Contributors. Debugging your program using Valgrind gdbserver and GDB, 2014. `http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver`.

[100] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. 2010.

[101] S. P. Reiss and M. Renieris. JOVE: Java as it happens. 2005.

[102] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. 2010.

[103] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. 2011.

[104] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: a large-scale study of the use of eval in JavaScript applications, 2011.

[105] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. 30(12):889–903, Dec. 2004.

[106] D. Röthisberger. Querying runtime information in the IDE. 2008.

[107] D. Röthliberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. 2009.

[108] D. Röthlisberger. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, Universität Bern, 2010.

[109] Y. Saito. Jockey: a user-space library for record-replay debugging. 2005.

[110] R. Salkeld, B. Cully, G. Lefebvre, W. Xu, A. Warfield, and G. Kiczales. Retroactive aspects: Programming in the past. 2011.

[111] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. 2013.

[112] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. 34:434–451, 2008.

[113] M.-A. Storey. Theories, methods, and tools in program comprehension: Past, present, and future. 2005.

[114] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. 1997.

[115] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: an interactive environment for information visualization and navigation. 2002.

[116] J. Stylos and B. A. Myers. Mica: a web-search tool for finding API components and examples. 2006.

[117] M. Taeumel, B. Steinert, and R. Hirschfeld. The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. 2012.

[118] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. 2003.

[119] E. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. 2010.

[120] E. Tanter, I. Figueroa, and N. Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations, and applications. *Sci. Comput. Programming*, 80:311–342, 2014.

[121] Telerik. Fiddler help: Decrypting HTTPS-protected traffic, 2013. `http://www.fiddler2.com/fiddler/help/httpsdecryption.asp`.

[122] Telerik. Fiddler web debugging proxy, 2013. `http://www.fiddler2.com/fiddler2/`.

[123] M. Terrano and P. Bettner. 1500 Archers on a 28.8: Network programming in Age of Empires and beyond. In *Gamasutra*, Mar. 2001. `http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php`.

[124] The Selenium Project. Selenium WebDriver documentation, 2012. `http://seleniumhq.org/docs/03_webdriver.html`.

[125] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. 2010.

[126] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. 1990.

[127] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. 2010.

[128] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *Commun. ACM*, 40:38–43, Apr. 1997.

[129] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. 2013.

[130] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: a universal reversible debugger based on decomposing debugging histories. 2011.

[131] I. VMWare. Replay debugging on Linux, Oct. 2009. `http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf`.

[132] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. 1993.

[133] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2002.

[134] WebKit contributors. The WebKit open source project, 2012. `http://www.webkit.org/`.

[135] WebKit contributors. About Safari Web Inspector, 2014. `https://developer.apple.com/library/safari/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html`.

[136] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. 30(2), 2005.

[137] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI screenshots for search and automation. 2009.

[138] K. Yit Phang, J. S. Foster, and M. Hicks. Expositor: scriptable time-travel debugging with first-class traces. 2013.

[139] S. Yoo, D. Binkley, and R. Eastman. Seeing is slicing: Observation based slicing of picture description languages. 2014.