Understanding Dynamic Software Behavior with Tools for Retroactive Investigation

BJ 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

Amy J. Ko, Chair

Steve Tanimoto

James Fogarty

Sean Munson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Understanding Dynamic Software Behavior
with Tools for Retroactive Investigation

BJ Burg

Co-Chairs of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science and Engineering
Associate Professor Amy J. Ko
The Information School

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 input events—such as user interactions, network callbacks, and asynchronous tasks; (2) by retroactively logging program states and reverting 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

I would like to thank Jan Vitek and Gregor Richards for taking a chance on me back when I was a sarcastic, inexperienced undergraduate at Purdue University. Without their enthusiasm for research, danger, and hacking, it's unlikely I would have chosen such a fun dissertation topic! A huge thanks to Filip Pizlo, Timothy Hatcher, and others who convinced me to contribute my replay prototypes back to the WebKit project; the experience has improved this work significantly, and may yet lead to big real-world impacts.

My studies at the University of Washington would have been short-lived if not for the patient, firm guidance of my co-advisors. Mike taught me the fundamentals of doing research back when I had big ideas and few concrete plans. Amy sold me on the vision of HCI and software engineering when I was at my most cynical, and helped to channel my energy and ideas into the compelling set of projects contained in this dissertation. I will miss both Amy and Mike's honest and open feedback—red marks and all.

Many other people have helped to shape my research in one way or another. My lab buddies—Ben, Todd, Colin, Sai, Kivanc, Ivan—kept me company in our windowless 3rd floor research den. I was fortunate to supervise the UI-building efforts of two wonderful undergraduate research assistants, Katie Madonna and Jake Bailey. Discussions with external researchers—Greg Wilson, Chris Parnin, Thomas LaToza, Adrian Kuhn, Joel Brandt, David Herman, Patrick Walton, Lars Bergstrom, Wolfram Schulte, among many others—helped me fine-tune my ideas and see the broader impacts of my research agenda. Outside of academia, my skill as a researcher and hacker leveled up each time I interned: once at Microsoft Research, twice at Mozilla Research, and once Apple Inc., where I will continue this line of work after graduation.

Lastly, I thank my friends and family for their support, friendship, and love during this long uphill climb. On campus, Katie, Conrad, Karl, Sonya, Katelin, Eric, Lillian, Nicola, Nick, Mark, Matt, Brandon(s), Will, Vincent, Ricardo, Igor, Adrian, Lindsay, Elise, Rebecca, Tracy, and many others have brightened my day countless times. At home, my wife Steph is my co-conspirator, the roof over my head, the floor I stand on, and the one who presses me to keep my hopes up and my cynic down. Thanks to my family for cheering me on and always supporting me regardless of the direction I venture.

BJ Burg

Mariposa, California

June 26, 2015

TABLE OF CONTENTS

Page

Abstract iii

Acknowledgments iv

Table of Contents ix

List of Figures x

List of Tables xii

Chapter 1: Introduction 1
1.1 The Problem 2
1.2 Addressing the Problem 3
1.3 Definitions 4
1.4 Contributions 6
1.5 Outline 7

Chapter 2: Related Work 9
2.1 Capturing Executions 9
2.1.1 Deterministic Replay 9
2.1.2 Post-mortem Approaches 11
2.1.3 Navigating Captured Executions 13
2.2 Extracting Program States 14
2.2.1 Specifying Instrumentation 14
2.2.2 Inserting Instrumentation 15
2.2.3 Composing and Scoping Instrumentation 17
2.2.4 Decoupling Execution and Instrumentation 18
2.3 Designing Developer Tools 18

2.3.1 Cognitive Models of Comprehension and Tool Use 19
2.3.2 Information Needs and Developers' Questions 19
2.4 Understanding Dynamic Behavior 20
2.4.1 Visualizing Dynamic Behaviors 21
2.4.2 Visualizing and Exploring Recordings and Traces 22
2.4.3 Supporting Behavior Dissemination 23
2.5 Feature Location 24
2.5.1 Locating Features using Visual Output 24
2.5.2 Explaining How Interactive Behaviors Work 24

Chapter 3: Deterministic Replay for Web Programs 26
3.1 Background 27
3.1.1 Web Programs 27
3.1.2 Rendering Engines 29
3.1.3 Features, Ports, and Platforms 29
3.1.4 Browser Architecture 30
3.2 Design 30
3.2.1 Types of Nondeterminism 31
3.2.2 Intercession Mechanisms 34
3.2.3 Recording and Input Structure 35
3.3 Implementation 35
3.3.1 Capturing and Replaying Executions 36
3.3.2 External Nondeterminism 37
3.3.3 Internal Nondeterminism 39
3.4 Evaluation 40
3.4.1 Fidelity 40
3.4.2 Performance 41
3.4.3 Scalability of the Approach 42
3.4.4 Limitations 43
3.5 Summary 45

Chapter 4: Replay Extensions and Applications 52
4.1 Navigating to Program States 53

4.1.1 Indexing Executed Statements . . . 53
4.1.2 Debugger Bookmarks . . . 54
4.2 Extracting Program States . . . 55
4.2.1 Scanning Algorithms . . . 55
4.2.2 Virtualizing State Extraction . . . 56
4.3 Mitigating Replay Faults . . . 58
4.4 Summary . . . 61

Chapter 5: Interfaces for Navigating Executions . . . 62
5.1 Basic Replay Controls . . . 63
5.2 Navigating via Inputs . . . 65
5.2.1 Interface Design . . . 65
5.2.2 Example . . . 67
5.2.3 Debugger Bookmarks . . . 69
5.2.4 Breakpoint Radar . . . 70
5.2.5 Tool Integration . . . 72
5.3 Navigating via Runtime States . . . 73
5.3.1 Interface . . . 75
5.3.2 Example . . . 76
5.4 Summary . . . 79

Chapter 6: Explaining Visual Changes in Web Interfaces . . . 80
6.1 Motivation . . . 80
6.2 Example . . . 82
6.3 Interface Design . . . 86
6.3.1 Design Rationale . . . 86
6.3.2 Tracking an Element . . . 89
6.3.3 Relating Outputs and States . . . 90
6.3.4 Comparing Internal States . . . 91
6.3.5 Finding Change Causes . . . 91
6.4 Implementation . . . 92
6.4.1 Detecting Visual States . . . 92
6.4.2 Capturing State Snapshots . . . 94

6.4.3 Comparing State Snapshots . . . 95
6.4.4 Explaining State Differences . . . 97
6.4.5 Instantiation . . . 101
6.5 Practical Experience with Scry . . . 101
6.5.1 Expanding Search Bar . . . 101
6.5.2 A Tetris Clone . . . 102
6.5.3 A Fancy Parallax Demo . . . 103
6.6 Limitations and Future Directions . . . 104
6.7 Summary . . . 105

Chapter 7: How Developers Use Timelapse . . . 107
7.1 Study Design . . . 107
7.2 Participants . . . 108
7.3 Programs and Tasks . . . 108
7.3.1 Space Invaders . . . 108
7.3.2 Colorpicker . . . 109
7.4 Procedure . . . 110
7.5 Data Collection and Analysis . . . 111
7.6 Results . . . 112
7.7 Discussion and Summary . . . 113

Chapter 8: Future Work . . . 115
8.1 Collaborative Debugging . . . 115
8.2 Creating Tests From Recordings . . . 117
8.2.1 User Interface Tests . . . 118
8.2.2 Performance Regression Testing . . . 118
8.2.3 Minimizing Recordings . . . 120
8.3 On-demand, Retroactive Dynamic Analysis . . . 122
8.3.1 Improving Scalability and Reliability . . . 123
8.3.2 Making Dynamic Analysis Interactive . . . 124
8.4 A Database of Reusable Executions . . . 125

Chapter 9: Conclusion . . . 128

Appendix A: Research Prototypes and Demos 130
A.1 Prototypes 130
A.1.1 Before Timelapse 130
A.1.2 `timelapse-hg` 130
A.1.3 `timelapse-git` 131
A.1.4 `timelapse-git` Redesign 134
A.1.5 `replay-staging` 134
A.1.6 `scry-staging` 135
A.2 Demos 135

LIST OF FIGURES

Figure Number — Page

1.1 Major parts and chapters of this dissertation. 7

3.1 Input specifications for GetCurrentTime and SetRandomSeed. 48
3.2 Input specification for DidReceiveData. 48
3.3 A `dispatch` implementation for DidReceiveData. 49
3.4 Code that interposes on the `new Date()` API. 49
3.5 Code that saves and restores the initial random seed. 50
3.6 Code that captures an input for incoming resource data. 51

4.1 Diagram of a scanning algorithm for extracting program states. 57
4.2 Proposed design for virtualized extraction of program states. 59

5.1 An overview of Timelapse's user interface. 66
5.2 The multiple-bullets bug in the Space Invaders game. 68
5.3 Timelapse's visualization of debugger status and breakpoint history. 71
5.4 Logging statements used to debug a failure in Space Invaders. 73
5.5 The probes interface while debugging DOMTris. 75
5.6 A rounding bug in the Colorpicker widget. 76
5.7 How to use data probes to debug the Colorpicker failure. 77
5.8 Probe samples that are useful for debugging the Colorpicker failure. 78

6.1 A picture mosaic widget used in Scry's opening case study. 83
6.2 An overview of the Scry workflow for localizing visual changes. 85
6.3 An overview of Scry's user interface. 87
6.4 Scry's interface for comparing visual state snapshots. 90
6.5 Scry's interface for showing the operations that were responsible for a change. 92

7.1 The Colorpicker widget. 109
7.2 A summary of task time and success per condition and task. 114

A.1 An early prototype of Timelapse's input-centric timeline visualization. . . . 132
A.2 A screenshot of the same visualization, with different toggled options. . . . 133

LIST OF TABLES

Table Number Page

3.1 Major sources of external nondeterminism in rendering engines. 33
3.2 Performance measurements for several representative web programs. 47
6.1 Input mutation operations as defined by Scry. 95
6.2 Possible cases for Scry's per-node change summaries. 97

Chapter 1

INTRODUCTION

The world wide web's rise as *the* universal runtime environment has democratized the development of documents, applications, and user interfaces. Unlike on most platforms, web programs are transmitted in source form as hypertext markup language (HTML), cascading style sheets (CSS), and JavaScript code. Using development tools included with most web browsers, anyone can live-inspect and modify a client-side web program's source code and runtime states. Such live inspection facilities can dramatically shorten a developer's feedback loop as they create or fine-tune a web program's visual elements and behaviors.

Despite the advantages provided by live inspection, it is often no easier to understand and debug programs written for the web platform than those written for any other platform. Modern web programs are complex, interactive applications built using multiple tools, frameworks, and languages. While originally intended to support hyperlinked documents, web technologies such as HTML, DOM, JavaScript and CSS have evolved over 20 years to become the building blocks for large, interactive, cross-platform applications. As a hodgepodge of accidentally mainstream technologies[1], the web platform suffers from significant incidental complexity. The combination of declarative styles and rendering with imperative JavaScript code works to obfuscate a web program's dependencies and causal relationships. In practice, the complexity of a web program—like any other program—is limited by a developer's ability to understand and maintain these systems with available developer tools.

---

[1]The first version of JavaScript was written in 10 days by Brendan Eich at Netscape [54].

1.1 The Problem

Existing developer tools are inadequate for understanding and debugging interactive behaviors in web programs. Several important tasks in this domain—such as reproducing complex interactions and behaviors, finding runtime states related to a behavior, and locating the code that implements a behavior or visual effect—are especially difficult. Underlying the difficulty of these tasks is the fundamental limitation that *execution can only proceed forwards*. As a result, existing tools allow a developer to inspect current or future program states in different executions, but not past program states from a single execution. To inspect a behavior using a tool, a developer must set up her tools—such as breakpoints or logging—before the behavior actually happens. This places a considerable "iteration tax" on debugging: to gather data about what happened, a developer must find relevant pieces of source code, set up her tools, and reproduce the behavior once per tool configuration. In the face of nondeterministic behavior, repeatedly reproducing the same behavior after changing tool configurations is error-prone or impossible.

The fundamental limitation of forward-only execution reduces the usefulness of existing tools such as debuggers, profilers, or logging, and reduces the tasks that new tools can hope to support. First, every tool use requires behavior reproduction, making a tool's output tightly coupled to a specific execution. Thus, tools that cause performance problems or halt an ongoing execution are infeasible to use while interactive behavior is being demonstrated. Second, tools must be configured preëmptively without live feedback. Most developer tools operate on specific lines of code, but a developer may not know which lines of code implement an interactive behavior until it has already occured. It often takes many iterations with some tools (i.e., breakpoint debugger) to find the right place to use other tools (i.e., logging statements). Finally, tools do not directly support navigating temporally backwards from causes to effects. A developer must observe effects, manually find possible causes, and gather additional data by setting up her tools earlier in new execution.

1.2 Addressing the Problem

To move beyond the current status quo of developer tools, tool creators must think beyond the current limitation of forward-only execution. What if it were possible to go "back in time" and revisit any past program state from a single execution? Suddenly, many of the conventions and restrictions embedded in the designs of traditional tools become irrelevant. An execution is now an enormous corpus of runtime states, and a developer tool can be used *retroactively* after the original execution finishes to extract data from this corpus. A debugger can step forward and backward; a breakpoint becomes a trace of control flow over time; a logging statement becomes a trace of runtime state over time; and a profiler aggregates control flow and timing data.

Consider the task of debugging an interaction in a video game: instead of manually reproducing game behavior whenever different runtime data is desired, a developer could play the game once. Then, she can go "back in time" at will to gather program states that help understand specific program behaviors. As her understanding of the program grows, she can quickly switch between retroactive developer tools without reproducing the entire behavior again. 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 searching for relevant source code to instrument.

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 web program executions and reproduce past program states; and how tools support a developer in finding past program states within a captured execution during program understanding tasks. Chapter 3 investigates how deterministic replay techniques can be used to capture executions of event-driven programs running on high-level managed runtimes, such as a web program executing in a browser rendering engine. Chapter 4 develops several programmatic interfaces for collecting and revisiting past program states from a captured execution. These interfaces serve as a crucial linkage between the low-level capabilities of a deterministic replay infrastructure and the data- and time-oriented needs of retroactive developer tools. In order for a developer to act upon past program states, it must be possible to quickly find relevant bits of information among the vast corpus of runtime data produced during an execution. Chapters 5 and 6 present new retroactive developer tools that support several task-oriented strategies for finding and navigating to relevant program states in a captured execution. Chapter 7 describes an exploratory user study that investigates how one of these retroactive tools is used by developers during representative debugging tasks.

1.3 Definitions

This dissertation builds upon work from various disciplines and fields such as Human-Computer Interaction (HCI), 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* is the consumer of software produced by a *developer*. 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 are themselves

developers.

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 engineering-related terms also deserve definition. A *fault* is a latent flaw in a system, such as an incorrect algorithm, design, or implementation. An *error* is an incorrect or unexpected internal program state that may lead to a failure. A *failure* is an undesirable program output that does not conform to the program's expected behavior. This dissertation uses the terms *fault* and *defect* interchangeably to refer to parts of program code that cause an *error* and/or *failure*. A *bug* loosely refers to single or a combination of failure(s), fault(s), and/or defect(s); usage of this vague term is generally avoided in this document.

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 term *web program*[3] to refer to any document consisting of 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.

---

[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*.

[3]In other contexts, web programs are also referred to as web pages, web content, web applications, and other terms to emphasize program characteristics such as complexity and interactivity. This document

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 event-driven programs in managed languages (Section 3.2).

- A successful instantation of these replay techniques within the WebKit rendering engine (Section 3.3).

- The first categorization and description of important sources of nondeterminism in a production web browser (Section 3.3).

- An algorithm for revisiting any executed statement within a captured execution (Section 4.1.1).

- Invariants and runtime techniques for detecting replay errors and failures (Section 4.3).

- A timeline visualization of a captured execution that support navigating via top-level input events (Section 5.2).

- An interface for retroactively logging runtime states and revisiting their originating execution instants (Section 5.3).

- The first user study examining the benefits, drawbacks, and design concerns for interactive record/replay user interfaces (Chapter 7).

---

uses the single term *web program* and modifies it as necessary to convey the intended population of programs.

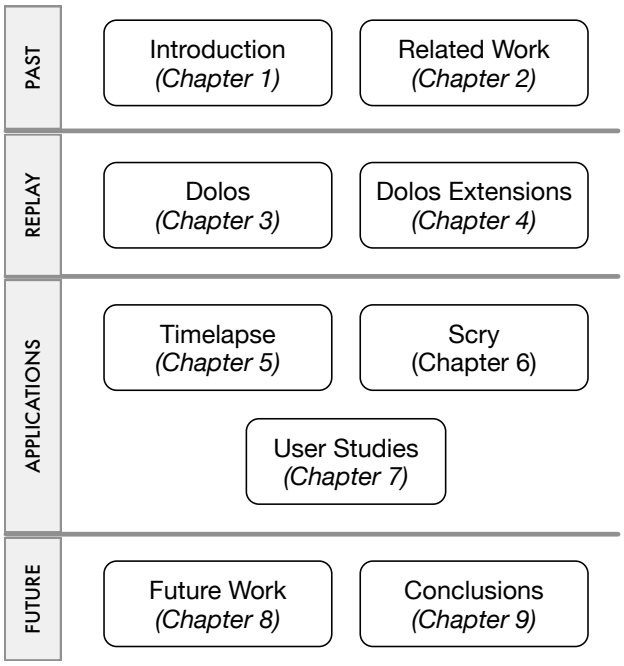

Figure 1.1: Major parts and chapters of this dissertation.

- Algorithms for efficiently detecting (Section 6.4.1), serializing (Section 6.4.2), and comparing (Section 6.4.3) visual states over time.

- An algorithm for establishing causality between visual changes, state changes, and JavaScript code (Section 6.4.4).

- An interface for feature location based on comparing output and state changes (Section 6.3.1).

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 in the two lines of research that this dissertation investigates: capturing executions and reproducing past program states; and tools

that aid a developer in finding and using past program states during program understanding tasks. The second part investigates a technical approach for capturing web program executions and extracting past program states from captured executions. Chapter 3 describes the Dolos deterministic replay infrastructure. Chapter 4 describes extensions to Dolos that support extracting past program states, and implement other important features used by retroactive developer tools. The third part investigates severals tools that aid a developer in finding and using past program states during program understanding tasks. It describes Timelapse and Scry, two program understanding tools that embody three different strategies for navigating captured recordings: via inputs (Section 5.2), via logged outputs (Section 5.3), and via visual state changes (Chapter 6). Chapter 7 presents the first user study that explores how replay interfaces are used during debugging tasks. The final part of this document sketches several directions for future research (Chapter 8) and presents the conclusions of this document (Chapter 9).

Chapter 2
RELATED WORK

In this chapter, I survey prior work in the two lines of research that this dissertation extends: capturing executions and extracting program states; and developer tools for finding and using program states during program understanding tasks.

2.1 Capturing Executions

The ability to revisit past program states within a single execution is a key enabler of the *retroactive* approach to program understanding as proposed by this dissertation. Since contemporary computer hardware only supports forward execution, the common approach to revisiting past program states is to *recreate* them indirectly. Instead of executing backwards, tools execute forwards in such a way that these states can be recreated.

Prior work that captures and recreates program states can be divided into two approaches: deterministic replay and post-mortem trace analysis. Tools based on deterministic replay recreate program states on-demand by exactly reëxecuting a captured program execution and extracting the desired live program state. Tools based on post-mortem analysis recreate program states by reconstructing them from a detailed trace of program operations collected during an execution.

2.1.1 Deterministic Replay

Deterministic replay is a widely-studied technique [41, 50] that recreates a specific execution by capturing and reusing all sources of nondeterminism that affect how the program executes. This dissertation focuses on the utility of deterministic replay for debugging nondeterministic systems [79, 113]. Deterministic replay has also been used to replicate

state between nodes in fault-tolerant systems [11, 22, 23, 26], replicate an entire execution to hot-backups [170], save executions for security auditing purposes [52], and other purposes. Research in the systems community focuses on deterministic replay of distributed systems [43, 85] and concurrent and parallel programs [37]; adding support for deterministic replay to hardware, virtual machines [52], operating systems [13], runtimes [10], languages [109, 151, 157], and applications [49, 65, 150, 162]. Others have explored novel applications of deterministic replay to new domains [34, 77, 118]. Few of these systems are widely used, and even the most robust commercial replay tools [113, 170] are designed for expert use via debugger commands (discussed in Section 2.1.3).

Some prior work exists that investigates the use of deterministic replay for replicating and debugging web program behavior. FireCrystal [123] is an extension for Firefox [112] that captures and replays DOM events on an isolated copy of the web program in order to recreate past visual states. WaRR [5] is a general-purpose web replay infrastructure based on WebKit; it captures and replays DOM events inside the rendering engine using hooks provided by the Selenium [163] plugin for WebKit. Mugshot [109] embodies a cross-platform, library-based approach to deterministic replay for web programs. It is packaged as a JavaScript library that can be injected into any running web program, and adds a simple playback interface overlay to the running web program.

"Language-level" deterministic replay—running a modified program on an unmodified operating system or application runtime, usually to achieve platform independence—is generally incompatible with the use of breakpoint debuggers and other privileged developer tools [122, 175]. One issue is that these approaches often modify a program using bytecode or source-to-source transformations (Section 2.2.2). Without a coordinating instrumentation framework (Section 2.2.3), these transformations are not composable and can accidentally change the target program's semantics or impact performance. A more fundamental limitation of this approach approaches is that they don't have access to all sources of nondeterminism that affect a sandboxed web program's execution. A sandboxed web program (using the GDB debugger's terminology [58], an *inferior process*)

cannot interact with privileged JavaScript debuggers (a *superior process*), so a replay system implemented in the inferior process is unusable while the breakpoint debugger has halted the program. 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.

Deterministic replay is sometimes *too* deterministic for certain use cases. Fully deterministic execution works against a developer if she wishes to reuse nondeterministic inputs on a different program version [34, 67, 77, 118] or intentionally diverge a replayed execution to explore alternate possibilities. Probes (Section 5.3) obviate the need to modify the program when inserting logging (the most common reason to alter a program during program understanding). Other work has investigated intentional divergence as an important use case. Scribe [88] is a multicore deterministic replay system designed to "go live" from the end of a captured execution; Dora [168] 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 [40], partial runtime state [75], or minimal replay logs [2]. By giving up on exact-fidelity determinism, these systems can re-execute more quickly by executing less code.

2.1.2 Post-mortem Approaches

The alternative to deterministic replay systems are post-mortem 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. This section discusses several trace-based systems that are designed for program understanding tasks.

Trace-based approaches to reconstructing program states have been used to support back-in-time debugging of x86 binaries.. Amber [120] and Nirvana [16] are two tools

for efficiently collecting, storing, and indexing traces from executions. Both tools use dynamic binary rewriting frameworks like Valgrind [117] 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 [97] 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.

Trace-based approaches have also been used for higher-level languages. The Omniscient Debugger for Java (ODB [101]) was the first trace-based omniscient debugger for Java programs. It heavily instruments the Java Virtual Machine (JVM) bytecode to record a trace of all memory activity and control flow. It uses information in the trace to populate views inside an integrated development interface (IDE) within value histories, rematerialized local variables, and a call stack. The omniscient debugger TOD [131] improves on ODB by incorporating modern database indexing and query optimization techniques and partial deterministic replay. STIQ [130] 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 [82] uses domain-specific instrumentation of user interface toolkits in order to save a trace of relevant toolkit invocations.

Trace-based techniques are generally avoided for JavaScript programs because the output of web programs is highly visual in nature and capturing a trace in memory can quickly make an application become unusably slow. Two exceptions are JSMeter [133] and DynJS [137], which both instrument the browser itself to collect a detailed trace of JavaScript execution for offline simulation.

2.1.3 *Navigating Captured Executions*

In order for deterministic replay to serve as the basis for retroactive developer tools, it must be possible to navigate to statements, events, and relevant program states within a captured execution. This section discusses different means for low-level navigation through captured executions.

Text-based commands are often the *only* interface for controlling back-in-time debuggers or deterministic replay infrastructures [113, 169, 170], and often supplement visual interfaces [101]. In 1990, Tolmach and Appel [165] first described the `reverse-step` and `reverse-continue` commands in a debugger for the ML language. Some seminal work for imperative debuggers is Boothe's description of efficient counter-based bi-directional debugging algorithms [21], which includes "reverse" versions of the debugger commands `step-into`, `step-out`, `continue`, and `watchpoint`. Arya et al. have recently proposed [7] algorithmic improvements to `reverse-watchpoint` based on decomposing large debugger commands [169] like `continue` into a sequence of `step-over` and `step-into` commands. These improvements are orthogonal to any specific replay or checkpointing strategy, as long as a stepping debugger operates similarly.

Timelapse's use of timelines and seekable outputs is specifically designed for casual use during program understanding tasks. It draws on a long tradition [69] of graphical history visualizations such as those used extensively in the Chronicle [62] tool. Few deterministic replay tools can seek execution directly to specific logged outputs without auxiliary use of breakpoints. The YingYang [107] 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 [77] combines interfaces 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 [82], TOD [131], and ODB [101] support navigating through traces by selecting simulated visual output, console output, or previous values of objects. Rather than using the selected output 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 Extracting Program States

This dissertation builds on the deterministic replay approach, which is dependent on additional methods for extracting program states from live executions produced on-demand. This section reviews important aspects of runtime data extraction: how to specify, implement, compose, and choose data-gathering instrumentation. Many of these aspects were first investigated in the context of dynamic analysis techniques [8] or aspect-oriented programming [78], but are equally applicable in other contexts.

2.2.1 Specifying Instrumentation

The ways in which data-collecting instrumentation are used varies widely between languages, tool chains, application domains, and use cases. Dynamic analysis techniques tend to contain the most sophisticated instrumentation techniques, because they primarily rely on runtime data [8]. Two defining characteristics of instrumentation are its *specification mechanism*: how instrumented source code and data is specified; and its *collection mechanism*: how a base program is modified to gather data. Specification mechanisms greatly affect the complexity of implementing a particular dynamic analyses or technique, while collection mechanisms greatly affect how instrumentation impacts performance.

There are a variety of declarative and imperative mechanisms for specifying what code or data should be instrumented. Aspects [78] have been used to declaratively specify instrumentation, especially for coarse-grained analyses in high-level languages such as Java [129, 131, 143] and JavaScript [100, 164]. However, aspects were not designed for instrumentation, so aspect languages have been extended to simplify common tasks like instrumenting basic blocks [47], sharing state across join points [105], and cheaply accessing static and dynamic context at runtime. For greater power, analyses must resort low-level bytecode manipulation libraries [12] to suit their needs. With any of these mechanisms, an analysis must not perform side-effecting operations on the instrumented program's data; this can make some analyses much more difficult to write.

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 [117] implements shadow registers and shadow memory for x86 binaries. Jalangi [148] 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 [106] is an asynchronous, isolated virtual machine (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 Inserting Instrumentation

Source instrumentation is the technique of instrumenting programs by inserting instrumentation directives directly into the program's source code. Source instrumentation is rarely used in traditional computing domains due to the lack of source code for arbitrary

program binaries. However, in the domain of web programs, programs are transmitted exclusively in source form and standardized bytecodes or binaries don't exist[1]. Thus, the vast majority of dynamic analysis tools for web programs [1, 108, 109, 121, 138, 148, 164] are implemented using source instrumentation. To instrument arbitrary web programs, these tools intercept incoming JavaScript sources using a reverse proxy [119]. They then parse and modify JavaScript sources using JavaScript abstract syntax tree (AST) libraries [66, 68, 114, 115], and pass the modified program to the browser.

Source instrumentation has severe drawbacks that make it unsuitable for use with existing developer tools that require a developer to view the resulting code. Transformed source code is effectively obfuscated, rendering the code jumbled in source code editors. A breakpoint debugger is of little use because it 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 execution level. Lastly, source instrumentation incurs high performance overhead during rewriting and at runtime, and is ill-equipped to handle the dynamic features of JavaScript [137] such as `eval` [139] and aliasing of native methods.

In most runtime environments, bytecode instrumentation [12, 82] and dynamic binary translation [16, 117] are the standard mechanisms for modifying programs for analysis purposes. Many dynamic analysis and instrumentation frameworks [12, 16, 105, 117, 148, 164] exist to reduce the engineering effort of instrumenting code at such a low level. 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. 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

[1]Recently, major browser vendors have convened the WebAssembly community group [172], with the goal of standardizing a compressed binary AST format for JavaScript code [27]. This proposal would reduce networking and parsing overhead, but would not simplify source instrumentation, which already operates at the AST level.

server [132] for GDB which translates debugger commands to work on instrumented code and additionally exposes values of shadow memory and shadow registers.

Bytecode-level instrumentation of JavaScript programs is relatively uncommon. Directly instrumenting a web browser's rendering engine or JavaScript runtime is inherently browser-specific, and requires much greater knowledge of the runtime environment. However, the approach has significantly better performance and can gather data from the virtual machine that is unavailable to JavaScript code [17, 137, 139].

2.2.3 Composing and Scoping Instrumentation

In order for multiple analyses to observe behaviors simultaneously, their respective instrumentation must be *composable* and not interfere with other analyses. Clearly, naively instrumenting low-level bytecode or source code is not composable because there is no way to distinguish instrumentation from client code. Researchers of aspect languages have long been concerned with unexpected interactions between aspects that use the same join points. Mechanisms such as stratified execution [159, 160] prevent aspects from advising other aspects. Ansaloni et al. [6] 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.

One important principle for reducing the space and runtime overhead of instrumentation is to only collect information that's actually needed. To limit the scope of instrumentation, prior work investigates the use of static and dynamic contexts to selectively instrument code or collect data, respectively. Reflex [158] supports spatial and temporal filters of behavioral reflection. Other tools adaptively add and remove dynamic instrumentation to running programs in response to user commands or execution events [20, 33, 126, 136]. However, most instrumentation frameworks (with the notable exception of DTrace [33]) 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.

2.2.4 Decoupling Execution and Instrumentation

Recently, researchers have investigated techniques for running a dynamic analysis "offline" by decoupling the instrumentation for an analysis from a live execution. This dissertation refers to the strategy of running dynamic analyses on a replayed execution [38, 146] as *retroactive analysis*. More common in the literature is the strategy of post-hoc trace querying [60, 120, 179] and analysis [96, 129, 130, 131, 133, 137], which this dissertation refers to as *post-mortem analysis* because access to a live execution is not necessary to perform the analysis. Several projects [16, 39, 130, 148, 179] 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.

2.3 Designing Developer Tools

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, human-computer interaction (HCI), ergonomics and computer science have developed several theories and models to account for program comprehension and tool use from a cognitive perspective. This section connects these traditional research results to more recent research that focuses on characterizing developers' information needs, and how these questions motivate comprehension tool research. Later, this section reviews related work

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 [48] provides a comprehensive history of cognitive models of program comprehension. Researchers originally modeled program 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 and Vans's integrated meta-model [171] is representative of influential cognitive models from the 1980's and early 1990's. Storey [152], Storey et al. [153] 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 [173] adapts the theory of distributed cognition [72] 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. [127] 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 [84, 93], collaborating, seeking information [25, 70],

describing implicit knowledge [35], and questions during software maintenance [149]. 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 [171] 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 [94, 95] and information-seeking [25, 83, 128, 149], and problem-solving strategies. Most relevant to this dissertation, researchers have catalogued common types of questions, including reachability questions [89, 92], hard-to-answer questions [90] and questions about output and causality [82]. These questions form a comprehensive account of a tool's capabilities from the perspective of its users; many tool papers (including this dissertation) devote a substantial amount of their motivation to considering how these questions could be answered by new capabilities and designs.

2.4 Understanding Dynamic Behavior

By providing appropriate user interfaces and using good visual encodings of data, developer tools can provide tremendous leverage during program understanding tasks. The remainder of this chapter reviews key aspects of developer tools, including common vi-

sual encodings of runtime behavior and some considerations for specific domains related to this dissertation.

2.4.1 Visualizing Dynamic Behaviors

Visualization is a large field with many applications to program comprehension. This section focuses 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.

At the lowest level, many tools expose dynamic behavior of specific expressions and statements by augmenting text editors with visualizations. These include popovers, background and foreground color shading [103], context menus [82, 144], inline gutter/scrollbar widgets [103, 144], runtime values [82, 103], or links to other views with information about specific instances [82, 101]. These 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.

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 [91] and sequence diagrams [82] are common ways of showing these Graphs and sequence diagrams are inherently distinct in form from source code, so visualizations must also include contextual hints (such as hyperlinks or a call stack) to remind the user of the context of 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 con-

veys relationships. For example, a horizontal timeline of multithreaded execution [166] instantly implies temporal dependencies among events generated by different threads, even if these are not necessarily accurate. A vertically-oriented call stack visualization [63] can exploit the convention of a stack growing downward to imply the relationship between callers and callees. The VIVIDE programming environment [156] uses an horizontal, infinitely-scrolling tape of connected editor windows to show past and current IDE views of a program. Lastly, the Code Canvas line of work [24, 45, 46] 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 conventions and can become cluttered. The Light Table IDE[2] 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 [154] were an early exploration of providing multiple levels of detail within the same editor. Other research has used multiple levels of detail to explain causality relationships for JavaScript events [1]. Some visualization tools specifically target discovery of high-level trends over the entire execution [44, 55, 74, 131, 142, 166]. Similarly, Röthisberger proposed [142] a query tool in the IDE to drive online partial dynamic instrumentation.

2.4.2 Visualizing and Exploring Recordings and Traces

In contrast to the dearth of interactive deterministic replay tools, there have been many tools [82, 131] to visualize, navigate, and explore execution traces[3] generated by an instrumented program. Execution traces are several orders of magnitude larger than recordings of nondeterminism and contain very low-level details. Thus, the only way to understand them is to use elaborate search, analysis, and visualization tools. While Timelapse visual-

[2]Light Table: `https://lighttable.com`

izes an execution as the temporal ordering of its inputs (Section 5.2), trace-based debugging tools [82, 166] 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 [82]).

Unfortunately, the practicality of many trace-based debugging tools is limited by their performance on modern hardware and the size of generated execution traces. Profilers, logging, tracing libraries [33] and other lightweight uses of instrumentation 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 [82, 120]. Generated traces and their indices [130, 131] are very large and often limited by the size of main memory.

2.4.3 Supporting Behavior Dissemination

Deterministic replay systems that support dissemination of behaviors have only been widely deployed as part of video game engines [49]. 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 [162]. In the wider software development community, bug reporting systems [59] and practices [184] emphasize 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 this dissertation focuses on the utility of deterministic replay systems for debugging, such systems are also useful for creating and evaluating software. Prior work has used capture/replay of real captured data to provide a consistent, interactive means

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

for prototyping sensor processing [34, 118] and computer vision [77] algorithms. More generally, macro-replay systems for reproducing user [163] and network [161] input are used for prototyping and testing web programs and other user interfaces. Dolos recordings (Chapter 3) 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 [138] uses this strategy to synthesize standalone web benchmarks. Derived inputs may improve the results of state-exploration tools such as Crawljax [108] by providing real, captured input traces.

2.5 Feature Location

2.5.1 Locating Features using Visual Output

Scry (Chapter 6) 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 programs [36], 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 [167], 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.

2.5.2 Explaining How Interactive Behaviors Work

Scry follows a long line of research [152] that aims to help a developer comprehend specific program features or behaviors. Recent work for user interfaces has focused on inferring behavioral models [1, 108, 110], logging and visualizing user inputs and runtime events [30, 123], and using program analysis to produce causal explanations of behaviors [82, 148]. Scry shares the same reverse-engineering goals as FireCrystal [123], which also logs and visualizes DOM mutations that occur in response to user interactions. How-

ever, 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 [82]. 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 [177] 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 [19, 180] 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.

Chapter 3

DETERMINISTIC REPLAY FOR WEB PROGRAMS[1]

Many program understanding questions that a developer might ask [80, 90, 149] can be answered using program states, but gathering program states is difficult given the limitations of current tools. As discussed in Section 1.1, developer tools are currently limited to inspecting current and future program states of multiple executions, rather than past states within a single execution. To gather past program states, a developer must configure her tool prior to the behavior's occurrence, reproduce the behavior with the tool enabled, and then finally obtain the desired program states from the tool. This tedious, multi-step iteration cycle requires her to repeatedly reproduce the program behavior she is inspecting, which can be time-consuming and error-prone [184]. 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 to easily reach a program state suitable for debugging the underlying fault.

Deterministic replay is a runtime technique that can capture a single program execution as it executes, and then automatically re-execute it repeatedly and automatically, without requiring manual user interaction. With deterministic replay, a developer would need to manually interact with the program just once. Using retroactive tools (Chapters 5 and 6) that automatically gather past program states, a developer could investigate program behavior without stopping to manually reproduce an interaction whenever past states were necessary.

Deterministic replay techniques have great potential, but have not been widely adopted. Prior deterministic replay systems for web programs (Section 2.1.1) have major shortcom-

[1]Contributions in this chapter are described in part in Burg et al. [30].

ings: they do not integrate well with breakpoints, logging, and other developer tools; they often have poor performance; and they do not have sufficient fidelity to replay complex web programs. The choice of deterministic replay over tracing for revisiting past program states is critical to the goal of enabling retroactive versions of existing developer tools, which generally operate on live executions rather that serialized program states. However, deterministic replay techniques cannot fulfill this vision if they are slow, exclude other tools, or cannot exactly recreate an execution.

This chapter describes Dolos, a novel deterministic replay infrastructure for web programs that addresses the shortcomings of prior work. 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—by implementing deterministic replay as a rendering engine feature. This chapter begins with some necessary background material (Section 3.1) and then introduces the design (Section 3.2) and instantiation (Section 3.3) of Dolos. The following chapter (starting on page 52) describes extensions to Dolos that support extracting program states, error detection, recording serialization, and other abilities that enable the retroactive developer tools described in Chapters 5 and 6.

3.1 Background

This section introduces important background concepts: what web programs are, how rendering engines execute web programs, how rendering engines and browsers are architected, and how this impacts deterministic replay.

3.1.1 Web Programs

Web programs are event-driven, interactive, and highly visual programs typically downloaded in source form over a network connection. Once its resources are parsed and evaluated, a web program's execution is driven by user input, asynchronous tasks, network

traffic, and other events. Interactions are programmed using JavaScript, an imperative, memory-safe, dynamically-typed scripting language. Web programs make extensive use of platform APIs to access persistent state, make network requests, programmatically render visual output.

The execution model of web programs mirrors that of typical graphical user interface (GUI) frameworks. Work performed by web programs is scheduled cooperatively in a single-threaded event loop, and execution is naturally divided into a series of *event loop turns*. An event loop processes each turn of work sequentially. Worker threads can perform parallel computation and communicate with the main program via message passing. A web program can communicate with other web program instances via message passing, but are otherwise isolated from other contexts.

As an evolution of a static document format, web programs do not have explicit boundaries of scope and extent like those associated with processes in modern operating systems. Web programs can embed sub-programs inside `<iframe>`, `<frame>`, and `<svg>` elements; the tree formed by transitively closing over this hierarchy of web programs is referred to as a program's *frame tree* (collectively, a *page*). The means of communication between parent and child programs in the frame tree depends on their origins. In some cases they can directly access each other's heap data, and in other cases they may only communicate via message passing. To simplify the situation for the purposes of deterministic replay, we consider the scope of a single execution to encompass an entire frame tree. An execution begins when the root node of the frame tree (hereafter, the *main frame*) initiates a navigation to a new document. An execution ends when the main frame initiates a navigation to a different document. Thus, the extent of a single execution is between two navigations of the main frame.

3.1.2 *Rendering Engines*

The core functionality of a web browser is referred to as a *rendering engine*. Rendering engines are complicated execution environments that produce a web program's visual output. A rendering engine processes inputs from the network, user, and timers; executes JavaScript; computes page layout; paints the document's visual representation into layers; composites multiple layers into a flat image, and renders the image to the screen. The rendering engine schedules asynchronous computation using a cooperative, single-threaded event loop. Features that are ancillary to a web program's execution, such as bookmarks and a browser's address bar, are provided by browser applications instead of the rendering engine.

3.1.3 *Features, Ports, and Platforms*

Rendering engines are usually designed to support multiple operating systems (hereafter, *platforms*), browsers, and application platforms (hereafter, *ports*). For example, the WebKit rendering engine is used by the Cocoa toolkit (Mac and iOS), GTK toolkit (Mac OS X, Windows, Linux), and EFL toolkit (Enlightenment); the Blink rendering engine is used by the Chrome browser (Mac OS X, Windows, Linux), Opera browser, and many other applications. Some rendering engine ports expose separate public APIs, which allow external programs (hereafter, *embedders*) to embed the rendering engine and customize its settings and behavior in predefined ways.

How a web program executes is highly dependent on the specific rendering engine used to execute it. To support a variety of uses, rendering engines aggressively modularize capabilities and features, allowing some features to be enabled, disabled, or customized at compile-time or runtime. Most JavaScript-accessible APIs are standardized, but many lack common test suites, leading to subtle interoperability issues. Many rendering engines expose nonstandard APIs, new input modalities, and embedder-specific functionality. To use available features without depending on them, web programs often

perform *feature detection*—programmatically testing for existence of specific features—and alter their execution based on their execution environment.

3.1.4 Browser Architecture

Modern browser architectures [135] are designed primarily with performance and security concerns in mind. Servo [4], WebKit, and Blink/Chromium use a multi-process model to enforce least privilege, isolate different web programs, and provide coarse-grained parallelism. Low-privilege tasks such as executing JavaScript, rendering/painting web content, graphics compositing, networking, tool interfaces, and persistent state storage run in their own child processes and communicate with a parent process via message-passing. Each child process is isolated using operating system sandboxing; if one child process crashes due to a failure, other processes are unaffected.

Multi-process architecture has several implications that make deterministic replay easier to achieve. Strict interfaces at process boundaries help to reveal potentially nondeterministic data flows between major browser and engine components. Messages between processes cannot reference shared mutable state, so they must fully and exactly characterize inputs which are often nondeterministic. A multi-process architecture also ensures that access to persistent state, network, and other nondeterministic external resources is virtualized, making it much easier to make these resources behave in a deterministic manner. Without a multi-process architecture, many of these invasive abstractions (strong interfaces, non-shared state, virtualized resources) must be reimplemented as prerequisites for deterministic replay.

3.2 Design

The remainder of this chapter describes the design and implementation of Dolos, a deterministic replay infrastructure for web programs. The primary purpose of Dolos is to enhance existing workflows (Chapter 5) and enable new developer tools (Chapter 6) by

making it possible to revisit past program states within a single execution.

Concretely, the design of Dolos supports these use cases with following requirements:

1. **Low overhead.** Recording must introduce minimal performance overhead, because many web programs are performance-sensitive. Replaying must be fast so that users can quickly revisit past program states.

2. **Exact re-execution.** Recordings must exactly reproduce observable web program behavior when replaying. Replaying should not have effects on the network, persistent state, or other external resources.

3. **Non-interference.** Deterministic replay must not interfere with the use of tools such as breakpoints, profilers, element inspectors, and logging. (Source-to-source instrumentation in particular is disallowed by this requirement.)

4. **Deployability.** It should be possible to casually use deterministic replay functionality without special hardware, installation, configuration, or elevated user privileges.

These requirements induce significant design constraints that have not been fully addressed by prior work (further discussed in Section 2.1.1). The closest points in the design space of deterministic replay techniques are those developed for operating systems [13] and virtual machines [52]. Like these systems, Dolos provides an execution environment on which arbitrary programs can be captured and replayed without modifications. Dolos must mediate access to nondeterministic resources and APIs while allowing nondeterministic and deterministic programs to execute side-by-side.

3.2.1 Types of Nondeterminism

Dolos achieves low overhead by effectively *virtualizing* sources of nondeterminism and otherwise executing a web program using the rendering engine's normal code paths.

From the rendering engine's point of view, the web program just so happens to make the same requests every time; from the web program's point of view, the rendering engine just so happens to behave the same way every time.

Compared to other execution environments [73, 113, 170], web program executions are easier to capture and replay in some aspects, and more difficult in others. The single-threaded execution model, cooperative scheduling, and memory safety of web programs make it unnecessary to record thread schedules, instruction counts, and low-level hardware/register states. On the other hand, the plethora of high-level client APIs, low-level platform APIs, modes of interaction, and complexities of retrofitting a virtual machine make it very difficult to find and address all sources of nondeterminism that affect execution.

Nondeterminism manifests in two ways as a web program executes. Table 3.1 provides an overview of sources of nondeterminism that are common to all web rendering engines. *Environmental inputs* are values returned by nondeterministic browser APIs as they are called by JavaScript code. Web programs use these APIs to detect device characteristics, access persistent storage, and interact with external resources. *Event loop inputs* are nondeterministic events that drive execution by evaluating new code, dispatching DOM events, or running JavaScript callbacks directly. Event loop tasks are received by the rendering engine, enqueued into the main event loop, and later executed. Most event loop inputs originate from outside of the rendering engine, and consume an entire event loop turn.

Nondeterminism originates from both internal and external sources [13]. In rendering engines, internal nondeterminism arises when the rendering engine itself needs to schedule event loop tasks; if the tasks can transitively cause JavaScript to execute, then the contents and ordering of these tasks with respect to other event loop inputs is a source of nondeterminism. Section 3.3.3 discusses some examples of internal nondeterminism encountered in the WebKit rendering engine.

External nondeterminism originates from outside of the rendering engine; in a multi-

Input	Classification	DOM Events & APIs
Keyboard strokes	Event Loop	`keyup`, `keypress`, `keydown`
Mouse input	Event Loop	`mouseover`, `click`
Scroll wheel	Event Loop	`scroll`, `mousewheel`
Page focus/blur	Event Loop	`focus`, `blur`
Window resize	Event Loop	`resize`
Document navigation	Event Loop	`unload`, `pagehide`
Timer callbacks	Event Loop	`setTimeout`, `Promise`
Asynchronous events	Event Loop	animation events
Network response	Event Loop	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	`document.cookie`
Policy decisions	Environment	`beforeunload`

Table 3.1: Major sources of external nondeterminism in rendering engines.

process browser architecture, these correspond to messages sent between the rendering process and other processes. External nondeterminism can manifest as both event loop inputs and environmental inputs. Most user interactions are examples of external nondeterminism that is entered into an event loop. When a user types characters, their browser forwards keystroke events to the rendering engine; the rendering engine queues the input event into its event loop and later acts upon the input when all prior inputs have been handled. Examples of environmental external nondeterminism include application-specific policies and persistent states. When a user clicks on a link, the rendering engine first asks the browser whether the proposed navigation is allowed by the browser's se-

curity policy before proceeding. Data storage for persistent state (cookies, local storage, etc.) is managed by the browser and accessed as needed by the rendering engine.

3.2.2 Intercession Mechanisms

Dolos uses three mechanisms to mediate sources of nondeterminism during capturing and replaying. The *capture/inject* mechanism intercepts event loop inputs when capturing an execution, and later injects the captured inputs during re-execution. The *save/restore* mechanism takes a snapshot of an initial state when capturing, and restores the state snapshot when replaying. *Memoization* works at the function call level to save and reuse the results of each invocation. Below, I describe how each of these is deployed to control common sources of nondeterminism in rendering engines.

Dolos uses capture/inject mechanisms exclusively to control event loop inputs. This mechanism has two responsibilities: to ensure the same computations are enqueued and processed by the rendering engine's event loop on capture and replay; and to prevent any "live" event loop inputs from being processed during playback. For example, if a user started typing as a web program is being replayed, the web program should not process the user's keyboard events because new interactions with the program would likely cause execution to diverge. Capture/inject mechanisms can intercept and inject event loop inputs either when they are enqueued or as they are processed. Dolos takes the latter approach for reasons described in Section 3.3.1.

Dolos uses both save/restore and memoization mechanisms to control environmental inputs. Save/restore can be used in cases where an initial state—such as a random number seed—can be cheaply and completely saved when capturing begins and restored when playback begins. Once the initial state is restored, subsequent calls to nondeterministic APIs based on this initial state do not need to be handled because execution is assumed to be deterministic. Memoization can also be used to control the same sources of nondeterminism by saving and reusing the values returned every time a related non-

deterministic API—such as `Math.random()`—is invoked by JavaScript code. In the case of random numbers, saving the random seed will always require saving less data with the same deterministic effect. In other cases, such as per-program persistent database stores, the memoization approach requires less space if the size of the initial state is large and the size of memoized values is small (and the nondeterministic function is invoked infrequently). In many cases, memoization is more straightforward to implement in existing rendering engines if there is not an existing mechanism for restoring an initial state.

3.2.3 Recording and Input Structure

Dolos recordings contain the data necessary to cause a deterministic execution. This data is organized hierarchically in a way that mirrors the structure of execution. At the top level, a recording session consists of one or more segments that correspond to a single web program execution, as defined in Section 3.1.1. Each segment is a self-contained web program, such that the execution it represents can be rearranged, added, or removed from a recording without affecting determinism of other segments[2]. At the next level of hierarchy, each segment contains a sequence of event loop inputs. The first elements of a segment typically represents the initial main frame navigation, actions to save/restore initial state, and then other event loop inputs as observed during capturing. At the lowest level of the hierarchy, each event loop input contains zero or more memoized inputs. Each memoized input belongs to an event loop input that corresponds to the event loop turn when the memoized input was saved or reused.

3.3 Implementation

Dolos instantiates the deterministic replay strategies outlined above in the context of WebKit [174], a popular rendering engine and browser toolkit. WebKit was chosen because

[2]This segmentation scheme is also useful as a crude checkpointing mechanism: since a segment does not depend on the execution of earlier segments, playback can begin from any segment. Checkpointing is further discussed in Section 3.4.4.

it contains the most widely-deployed rendering engine (WebCore) and web developer tools (Web Inspector). Dolos is implemented by modifications to WebKit's C++ and JavaScript codebase, and can be used as a drop-in rendering engine replacement for Safari by adjusting the dynamic library load path. This section describes implementation details that are specific to the instantiation of Dolos within WebKit. Many of the implementation choices described in this section may apply to other browsers and replay systems, but none are essential to the basic design of the deterministic replay infrastructure outlined above. These implementation strategies have been refined through multiple prototypes; The design that this chapter describes is realized by the `replay-staging` prototype (Section A.1.5). Each intercession mechanism and component of the capture/replay infrastructure has been redesigned several times. Appendix A provides some details about previous designs that were altered or abandoned.

3.3.1 Capturing and Replaying Executions

Dolos' strategy for capturing event loop inputs is to save them after they are dequeued from the event loop and before they are executed, rather than as they are enqueued into the event loop. Correspondingly, during playback Dolos recreates the actions of event loop inputs by re-executing (instead of re-enqueuing) them in order. In effect, Dolos captures the subset of the work performed in the rendering engine's event loop that can possibly run JavaScript code or impact the program's determinism. During playback, that same subset of work is initiated by Dolos without going through the event loop. The main benefit of this scheme is that executing event loop inputs is synchronous and only spans a single event loop turn. This makes certain functionality—such as pausing playback, aborting playback, or detecting unexpected execution—much easier to implement. Alternatives that require asynchronous event loop simulation, such as re-enqueuing event loop inputs, significantly complicate the replay engine's internal state machine and provide few benefits in return.

For any given event loop input, there may be multiple levels at which one could capture and inject the event. For example, when a user clicks the mouse, a series of steps occur: first, a `handleMousePress` message is sent to rendering engine from the parent process; second, the user input event is hit-tested against the view hierarchy to find potential event targets; third, the user input is translated into multiple DOM events such as `mousedown`, `click`, `dblclick`, or `dragstart`, depending on the target element and prior inputs; finally, each DOM event is dispatched to elements within the DOM tree, which may cause registered JavaScript event handlers to execute.

Unlike other some prior work that explores deterministic replay, Dolos attempts to capture event loop input events as early as possible in the processing pipeline outlined above. Mugshot [109] and other tools that rely on source instrumentation [123, 138] typically capture inputs later in the pipeline, either prior to hit-testing[3] or when inputs events are interpreted and dispatched as DOM events. Capturing DOM events is more difficult because it requires serializing event data and the element in the DOM tree to which the element was dispatched. This approach also has lower execution fidelity: rendering engines can perform arbitrary actions prior to dispatching related DOM events, such as moving form field focus, interacting with an input method editor (IME), or other state changes. By contrast, the strategy Dolos uses requires minimal memory use, introduces minimal runtime overhead, and is easy to implement because it copies simple, uninterpreted data structures before they are processed by the rendering engine pipeline. There is no need for Dolos to serialize the event target's position within the document tree because hit testing is deterministic.

3.3.2 External Nondeterminism

Dolos captures user input events and navigation events as they are sent to the rendering process using a capture/inject mechanism. Each message is saved to the active recording

[3]Guo et al. [65] report on the space and time benefits of memoizing application-level API calls instead of low-level system calls [145].

segment when capturing; during playback, each message is redelivered to the rendering engine through the same code paths as the original message. "Live" messages during playback, such as a user typing, are not delivered to the rendering engine to avoid divergence. WebKit uses platform-provided event loop implementations to handle incoming messages from multiple processes. Thus, the exact interleavings that Dolos records are determined in part by the underlying event loop implementation, and are generally not deterministic.

Dolos captures network traffic similarly to how it captures user input events. In WebKit, external resources—such as HTML, JavaScript, and images—are loaded asynchronously and in parallel. As each resource is downloaded, the networking process sends status update messages to the rendering process. The rendering engine uses these messages to generate placeholders, dispatch DOM events, or parse and run new JavaScript code. When capturing, Dolos saves these status update messages, their HTTP headers, and associated raw data. When replaying, Dolos silently blocks the web program's network requests from reaching the network process, and redelivers status update messages to simulate real network traffic. For example, when loading images asynchronously on Flickr, Dolos saves images as opaque byte buffers split across multiple "data received" status update messages. When replaying, Dolos redelivers status update messages with saved image data, and never communicates with Flickr servers.

The DOM provides several mechanisms that allow web programs to schedule asynchronous work, such as `window.setTimeout()`, `window.requestAnimationFrame()`, and the `Promise` API. Similar to other event loop inputs described above, Dolos captures and injects these callbacks as they are executed. During playback, Dolos directly executes callbacks in the observed order (without using the event loop) and prevents new callbacks from being scheduled during playback.

Environmental sources of nondeterminism are handled using of memoization and save/restore mechanisms, according to the implementation complexity and space requirements of each method. At the time of writing, most DOM APIs, such as `window.navigator`,

`window.screenX`, `window.localStorage` and `window.cookie`, are handled by adding memoization to the code that marshals data between JavaScript and C++. Cookies are an instructive case for whether to memoize or save/restore nondeterministic values. Most calls to `window.cookie` return the same value every time, suggesting that memoization wastes space. However, at the time of writing, cookie storage is handled by the browser instead of the rendering engine, so saving and restoring cookie state from within the rendering engine would require significant refactoring. Since repeated values in a recording can be interned (when stored in-memory) or compressed (when serialized), using memoization to handle cookies does not cause a significant increase in space usage.

For environmental nondeterminism handled through memoization, Dolos intercepts calls from the rendering engine to the underlying nondeterministic platform or port APIs instead of 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 its implementation of the `Date.now()` function.

3.3.3 Internal Nondeterminism

Because Dolos ensures full determinism of JavaScript, it must take an expansive, pessimistic view of what constitutes nondeterminism. Since simply executing JavaScript can cause divergence, Dolos must mediate all code paths that directly execute JavaScript or indirectly dispatch DOM events (and thus giving event handlers written in JavaScript a chance to execute). WebKit's rendering engine often uses single-shot timers to enforce asynchronous dispatching of DOM events. For example, `load` and `error` DOM events are dispatched asynchronously in the main event loop after an image or stylesheet is successfully parsed. Internally, WebKit uses asynchronous timers to defer the event dispatch. Dolos handles these internal asynchronous mechanisms in the same way that it controls public APIs for scheduling asynchronous work in the browser event loop.

3.4 Evaluation

To the best of our knowledge, Dolos is the first deterministic replay infrastructure that is deeply integrated into a rendering engine. With any new approach, we must wonder: is this way any better? Is replay faster or more correct? Is it more work to implement? What are its fundamental and incidental limitations? This section characterizes the Dolos approach to replay in terms of performance, replay fidelity, and scalability. The focus here is on the technical aspects of replay; later chapters demonstrate how Dolos can be extended (Chapter 4) and used as the basis for retroactive developer tools (Chapters 5 and 6).

3.4.1 Fidelity

The Dolos infrastructure attempts to reproduce[4] identical, deterministic JavaScript executions when capturing and replaying. There is no guarantee regarding number of layout reflows or paints due to time compression or internal rendering engine nondeterminism. The determinism of these computations is unimportant because visual output cannot affect the determinism of JavaScript computation. It is possible for JavaScript code to synchronously query the results of computed layout. Methods that perform such queries, such as `Element.offsetLeft`, implicitly suspend all other parsing and JavaScript execution until layout results have been updated. Thus, several layout runs may be coalesced, but results will always appear deterministic from the point of view of JavaScript code. Painting and graphics compositing are not observable from client-side JavaScript, and thus are similarly not addressed.

3.4.2 Performance

In my experience over the past 4 years, Dolos scales well to real-world interactive web programs without significantly impacting the user experience. Dolos only saves nondeterministic inputs when capturing an execution, so the recordings are very small and can be easily transferred and replayed on other computers. This section provides a historical snapshot[5] of Dolos' performance and maintainability characteristics. Subsequent prototypes (Appendix A) use the same architecture for capturing and replaying, but may be somewhat faster because the WebKit rendering engine continues to receive general performance optimizations.

Little effort has been spent thus far to optimize the replay infrastructure; despite this, capture and replay have negligible performance impacts. Table 3.2 describes performance characteristics of Dolos in a variety of modes and sample web programs. All numbers report the geometric mean of 10 runs (except for interactive runs, which were recorded once but replayed 10 times). The standard deviation (computed via arithmetic mean) was always less than 10% of the geometric mean. Dolos cleared network resource caches between executions to avoid memory and disk cache nondeterminism. In measurements of execution time, local copies of benchmarks were used to avoid nondeterminism caused by network latency and contention. Recording overhead and the amount of data collected scales with user events, network responses, and uses of environmental inputs, not CPU time.

Recording has almost no time overhead: execution times are dominated by the subject program. 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. Replaying at $1\times$ speed is marginally slower than a

[4]Section 4.3 describes several runtime mechanisms that can detect when and where execution has diverged.

[5]All measurements and numbers in this section were obtained in Spring 2013 using the `timelapse-git` prototype, as described in Section A.1.3.

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 user studies (Chapter 7), users attempted to minimize recording length to reduce the number of inputs that must be later searched, and rarely created recordings that exceeded a couple of minutes.

3.4.3 Scalability of the Approach

Dolos's design scales to new platforms, ports, and sources of nondeterminism, and represents a tiny addition to the rendering engine's codebase. The `timelapse-git` prototype of Dolos (Section A.1.3) consists of 7.6K source lines of code (SLOC), distributed across 74 new files and 75 modified files. For comparison, WebKit contains about 1.38M SLOC. Intercession mechanisms are typically installed at existing module boundaries, such as within the cross-language bindings between DOM and JavaScript, or at the rendering engine's process boundaries. Implementing Dolos required minimal changes to WebKit's architecture. Cases where clear boundaries already existed—such as user inputs, network traffic, and random numbers—made it easy to deploy intercession mechanisms. More substantial efforts were required in cases that lacked these boundaries, such as splitting execution into segments and handling internal nondeterminism.

At the code level, I have spent a lot of time to reduce the amount of code necessary to represent and control sources of nondeterminism. The name, type, and data members for each nondeterministic input are described declaratively in a JSON specification file; the build system automatically generates most "boilerplate" C++ code that is not unique to any input. Figures 3.1 and 3.2 show these specifications for the current time (memoized), random number seed (save/restore), and resource data received (event loop) nondeter-

ministic inputs. For each event loop input, Dolos additionally requires a hand-written `EventLoopInput::dispatch` method that encapsulates actions to take during playback to simulate an event loop input (Figure 3.3). Lastly, for each input, Dolos must insert an intercession mechanism into existing code. Figures 3.4, 3.5 and 3.6 show example uses of these intercession mechanisms for the same inputs listed above.

3.4.4 Limitations

Dolos is the first deterministic replay infrastructure for the web that attempts to be completely deterministic. This design goal is the fundamental reason why deterministic replay is a suitable foundation for retroactive developer tools (Chapter 4, Chapter 5, Chapter 8). It is also a very difficult goal to achieve and has been the source of many compromises and tradeoffs. This section describes some of the limitations inherent to the fully-deterministic playback approach taken by Dolos.

Limited Checkpointing

Unlike most prior research into deterministic replay, Dolos explicitly does not try to create intermediate checkpoints to accelerate random-access seeking times. However, Dolos can begin playback from any segment in a recording session, so this acts as a coarse-grained checkpointing mechanism. In initial user studies (Chapter 7), web program executions were usually short enough (and playback fast enough) that always replaying from the beginning was not too burdensome when using input and output-oriented navigation.

For longer recordings, or in situations where fast feedback is desired—such as "step-backward" debugger commands (Section 2.1.3)—checkpoints would be much more useful. Prior work has investigated client-side state migration [104] and snapshotting the JavaScript heap at the virtual machine (VM) level [10]. Neither of these approaches is directly applicable to Dolos. Neither addresses how to control the rendering engine's internal nondeterminism (Section 3.3.3), or how to serialize internal rendering engine states

that affect web program execution. Imagen [104] relies on source instrumentation and requires interposition on key DOM APIs, which can break existing web content, render debuggers useless, and prevent JavaScript engines from performing optimizations. Tardis [10] serializes the JavaScript heap efficiently, but does not save the state of the DOM. Further research is needed to efficiently create intra-execution checkpoints that can be used by Dolos.

Scope of Determinism

Dolos only ensures deterministic execution for *client-side* portions of web programs. 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 [181] 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. As plugins are sandboxed in their own process and interact with the rendering engine via well-defined APIs, Dolos could capture and reproduce the effects of a plugin on the web program using capture/inject and memoization mechanisms. During playback, the rendering engine would not be able to actually load and run the plugin content, since it may perform arbitrary computation.

The Dolos prototype does not address all known sources of nondeterminism[6], 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 mechanisms described in Section 3.2.2. Each new program input requires local changes to route control flow through a mechanisms and new code to marshall the input's data. Event loop inputs additionally require code to inject the input event during playback. The design documentation for the web replay feature [29] tracks handled and unhandled sources

of nondeterminism in mainline WebKit.

Rendering engines provide many APIs that are can be called nondeterministically by browsers, but do not affect web program determinism. For example, WebKit's rendering engine includes APIs for usability features like native spell-checking, in-page search, and accessibility. Dolos does not attempt to control these nondeterministic APIs because these features do not affect the execution of the web program, and a developer may wish to use such features differently during playback.

Generalizability

Dolos's record/replay strategy relies on having good places to virtualize sources of nondeterminism. Rendering engines or execution environments without clear boundaries will require greater engineering efforts to support deterministic replay. For example, the Gecko rendering engine used by the Firefox browser [112] is architected as dozens of decoupled components whose instances are shared between multiple web pages. Gecko is also a primarily single-process rendering engine. This design makes it easy to extend the browser and rendering engine, but difficult to capture and replay a specific web program in isolation from other web programs. Placing intercession mechanisms is also more difficult, as there are fewer existing boundaries between the browser and rendering engine.

3.5 Summary

This chapter adapts classical deterministic replay techniques to the domain of web programs: visual, event-driven programs written in high-level managed languages. Dolos is one instantiation of this approach for the WebKit rendering engine. Together, the design and instantiation of Dolos make the following contributions:

1. A software architecture for integrating deterministic replay into modern browser

[6] A somewhat outdated audit of nondeterministic APIs is available on at the following address: `https://github.com/burg/timelapse/wiki/Note-sources-of-nondeterminism`

rendering engines.

2. An enumeration and categorization of the sources of nondeterminism that impact how a web program executes.

3. Case studies illustrating how three mechanisms can efficiently interpose on important classes of nondeterministic inputs.

4. Evidence that deterministic replay has negligible space and time overheads for interactive web programs.

Program			Run Time						Data Size (KB)	
Name	**Description**	**Workload**	**Bottleneck**	**Baseline**	**Disabled**	**Recording**	**Replaying**	**Seeking**	**Log**	**Site**
JSLinux	x86 emulator	Run until login	network	10.5s	1.00×	1.65×	1.65×	0.37×	24.7 / 71.4 / 8.5	4500
JS Raytracer	ray-tracer	Complete run	CPU	6.3s	1.00×	1.01×	1.17×	1.02×	24.0 / 69.4 / 10.2	5.9
Space Invaders	video game	Scripted gameplay	timers	25.8s	1.00×	1.03×	1.22×	0.25×	247 / 683 / 56.8	712
Mozilla.org	home page	Read latest news	user	22.3s	1.00×	1.00×	1.09×	0.23×	187 / 502 / 29.5	2800
CodeMirror	text editor	Edit a document	user	16.6s	1.00×	1.00×	1.03×	0.07×	57.6 / 163 / 17.1	168
Colorpicker	jQuery widget	Reproduce defect	user	15.3s	1.00×	1.00×	1.07×	0.13×	112 / 302 / 26.7	577
DuckDuckGo	search engine	Browse results	user	14.1s	1.00×	1.00×	1.08×	0.19×	119 / 309 / 29.6	1900

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.

```
"inputs": {
    "JavaScriptCore": [
        {
            "name": "GetCurrentTime",
            "description": "Supplies the system time to Date.now() and new Date().",
            "queue": "SCRIPT_MEMOIZED",
            "members": [
                { "name": "currentTime", "type": "double" }
            ]
        },
        {
            "name": "SetRandomSeed",
            "description": "Sets the PRNG seed used by Math.random().",
            "queue": "SCRIPT_MEMOIZED",
            "members": [
                { "name": "randomSeed", "type": "uint64_t" }
            ]
        }
    ]
}
```

Figure 3.1: Input specifications for GetCurrentTime and SetRandomSeed. Each input specification lists the input's name, intercession mechanism type, and data members.

```
        {
            "name": "ResourceLoaderDidReceiveData",
            "description": "A resource loader received some data.",
            "queue": "EVENT_LOOP",
            "members": [
                { "name": "ordinal", "type": "uint64_t" },
                { "name": "frameIndex", "type": "uint32_t" },
                { "name": "buffer", "type": "SharedBuffer" },
                { "name": "encodedLength", "type": "int32_t" }
            ]
        }
```

Figure 3.2: Input specification for DidReceiveData. Input specifications can reference arbitrary C++ data types in describing data members, provided that serialization routines for the data type have been implemented.

```
109  void ResourceLoaderDidReceiveData::dispatch(Page& page)
110  {
111      if (ResourceLoader* loader = resourceLoaderForOrdinal(page, ordinal(), frameIndex()))
112          loader->didReceiveData(buffer()->data(), buffer()->size(), encodedLength(), DataPayloadBytes);
113      // FIXME: signal error if the callback couldn't be fired.
114  }
```

Figure 3.3: A `dispatch` implementation for DidReceiveData. Dispatching can access any internal rendering engine state, and must complete synchronously. This method finds an appropriate resource and synthesizes a callback using data from a buffer.

```
75  #if ENABLE(WEB_REPLAY)
76  static double deterministicCurrentTime(JSGlobalObject* globalObject)
77  {
78      double currentTime = jsCurrentTime();
79      InputCursor& cursor = globalObject->inputCursor();
80      if (cursor.isCapturing())
81          cursor.appendInput<GetCurrentTime>(currentTime);
82
83      if (cursor.isReplaying()) {
84          if (GetCurrentTime* input = cursor.fetchInput<GetCurrentTime>())
85              currentTime = input->currentTime();
86      }
87      return currentTime;
88  }
89  #endif
90
91  #if ENABLE(WEB_REPLAY)
92  #define NORMAL_OR_DETERMINISTIC_FUNCTION(a, b) (b)
93  #else
94  #define NORMAL_OR_DETERMINISTIC_FUNCTION(a, b) (a)
95  #endif
...
116 // ECMA 15.9.3
117 JSObject* constructDate(ExecState* exec, JSGlobalObject* globalObject, const ArgList& args)
118 {
119     VM& vm = exec->vm();
120     int numArgs = args.size();
121
122     double value;
123
124     if (numArgs == 0) // new Date() ECMA 15.9.3.3
125         value = NORMAL_OR_DETERMINISTIC_FUNCTION(jsCurrentTime(), deterministicCurrentTime(globalObject));
126     else if (numArgs == 1) {
127         if (args.at(0).inherits(DateInstance::info()))
128             value = asDateInstance(args.at(0))->internalNumber();
129         else {
```

Figure 3.4: Code that interposes on the `new Date()` API. The top half saves or fetches a memoized timestamp from the current recording. The bottom half shows the full implementation of `Date.[[Constructor]]`; replay-related code is highlighted in blue.

```
815 #if ENABLE(WEB_REPLAY)
816 void JSGlobalObject::setInputCursor(PassRefPtr<InputCursor> prpCursor)
817 {
818     m_inputCursor = prpCursor;
819     ASSERT(m_inputCursor);
820
821     InputCursor& cursor = inputCursor();
822     // Save or set the random seed. This performed here rather than the constructor
823     // to avoid threading the input cursor through all the abstraction layers.
824     if (cursor.isCapturing())
825         cursor.appendInput<SetRandomSeed>(m_weakRandom.seedUnsafe());
826     else if (cursor.isReplaying()) {
827         if (SetRandomSeed* input = cursor.fetchInput<SetRandomSeed>())
828             m_weakRandom.initializeSeed(static_cast<unsigned>(input->randomSeed()));
829     }
830 }
831 #endif
```

Figure 3.5: Code that saves and restores the initial random seed. This method is called whenever a new web program execution begins (i.e., a top-level document or iframe).

```
295  void SubresourceLoader::didReceiveDataOrBuffer(const char* data, int length, PassRefPtr<SharedBuffer> prpBuffer,
296  {
297      if (m_resource->response().httpStatusCode() >= 400 && !m_resource->shouldIgnoreHTTPStatusCodeErrors())
298          return;
299      ASSERT(!m_resource->resourceToRevalidate());
300      ASSERT(!m_resource->errorOccurred());
301      ASSERT(m_state == Initialized);
302      // Reference the object in this method since the additional processing can do
303      // anything including removing the last reference to this object; one example of this is 3266216.
304      Ref<SubresourceLoader> protect(*this);
305      RefPtr<SharedBuffer> buffer = prpBuffer;
306
307  #if ENABLE(WEB_REPLAY)
308      InputCursor* cursor = activeInputCursor();
309      if (cursor && cursor->isCapturing()) {
310          RefPtr<SharedBuffer> serializedBuffer = buffer.copyRef();
311          if (!serializedBuffer)
312              serializedBuffer = SharedBuffer::create(data, length);
313          cursor->appendInput<ResourceLoaderDidReceiveData>(ordinalForIdentifier(), frameIndexFromFrame(m_frame.ge
314      }
315      // If the main resource changes, then a new input cursor will be created for a new replay session segment.
316      RefPtr<InputCursor> protectCursor(cursor);
317      EventLoopInputExtent extent(cursor);
318  #endif
319
320      ResourceLoader::didReceiveDataOrBuffer(data, length, buffer, encodedDataLength, dataPayloadType);
321
322      if (!m_loadingMultipartContent) {
323          if (auto* resourceData = this->resourceData())
324              m_resource->addDataBuffer(*resourceData);
325          else
326              m_resource->addData(buffer ? buffer->data() : data, buffer ? buffer->size() : length);
327      }
328  }
```

Figure 3.6: Code that captures an input for incoming resource data. Code within the `ENABLE(WEB_REPLAY)` guard captures the relevant data for the callback and creates an event loop input with an isolated copy of the received resource data. The corresponding `dispatch` method executed during playback (Figure 3.3) calls this method again using the saved data.

Chapter 4

REPLAY EXTENSIONS AND APPLICATIONS[1]

Retroactive developer tools are distinguished by their ability to go "back in time" to reveal past program states. This ability is really the combination of two distinct capabilities: the capability to capture the essence of a specific execution, and the capability to extract and revisit program states within a captured execution. The Dolos deterministic replay infrastructure (Chapter 3) fulfills the former capability. This chapter describes extensions[2] to Dolos that fulfill the latter capability by supporting navigation within a captured execution and automated extraction of program states.

This chapter also describes other ancillary extensions to Dolos that support robust, fault-tolerant deterministic replay. While Dolos is designed to ensure completely deterministic replay, this is a top-level quality goal for its instantiations, rather than a guarantee of the Dolos design. Web browsers are large, complex pieces of software with many input and output modalities that involve nondeterminism. Thus, a deterministic replay infrastructure for such a system is likely to contain a large number of known and unknown faults where replay support is inadequate. Automatically detecting errors and handling any resulting failures is critical when using replay systems in real-world situations. These capabilities are also generally useful for finding and diagnosing faults in the Dolos infrastructure.

---

[1]The results in this chapter appear in part in Burg et al. [30] and Burg et al. [31]. Some features were upstreamed to WebKit and shipped in Safari 8; see Section A.1.5 for details.

[2]These additional features are referred to as *extensions* because they do not impact the architecture or intercession mechanisms used by Dolos. However, these features are not dynamic extensions [99] that can be toggled; rather, they are extra features added to Dolos itself.

4.1 Navigating to Program States

In order for a developer to jump back in time to specific values or statements, it must be possible to uniquely identify any executed statement and re-execute back to a given statement using commands provided by Dolos and a breakpoint debugger. This section introduces *timepoints* as the key abstraction for referring to a specific execution of a statement. Timepoints are used by Timelapse to implement debugger bookmarks (Section 4.1.2) and time-indexed outputs (Section 5.3); these uses are described in the relevant chapters.

4.1.1 Uniquely Identifying Executed Statements

A timepoint is a unique temporal address for a single execution of a statement (henceforth, an *execution instant*). Timepoints are a tuple consisting of four pieces of data: the current recording segment's offset, the preceding event loop input's offset, the source code location (file, line number, and column number), and an execution ordinal. The first two elements identify a specific event loop turn in which the timepoint occurred; the second two elements identify a single statement and a single execution of that statement. All of this data is readily available during capturing and playback except for the execution ordinal. An execution instant's ordinal can be obtained in two ways: by associating an increment-on-execute counter with each statement, or by counting breakpoint pauses at a statement since the previous event loop turn. Counters are automatically installed for statements that are known to be output-producing, such as `console.log` and `document.write`; counters for arbitrary statements can be added as-needed[3].

Dolos re-executes up to a timepoint in three phases. First, Dolos uses the recording segment offset and event loop input offset to replay to the preceding event loop input. Then, Dolos installs an increment-on-execute counter (described below) just prior the timepoint's statement; finally, execution is suspended by the debugger when the counter

[3]There is no conceptual reason why counters could not be maintained for every statement or basic block in the web program. These restrictions reflect the performance limitations of the current breakpoint-based implementation described below.

is incremented n times.

Counters are currently implemented using JavaScript breakpoint actions that are hidden from the user interface. Emitting these counters directly into the program's bytecode would permit much faster execution if the counter must be incremented many times, with the tradeoff of greater engineering effort. While breakpoints are a useful mechanism for suspending execution and collecting infrequent data, their use incurs a significant ($10\times$) performance overhead[4]. This slowdown can negate the interactive qualities of retroactive tools, which may lead a developer to use them in a more cumbersome batch-oriented manner. In its current form, Dolos uses several strategies to minimize the use of breakpoints. When replaying to a timepoint, Dolos completely disables the JavaScript debugger until playback has reached the preceding event loop input. Then, Dolos installs a counter by setting a single breakpoint at the timepoint's statement. This limits breakpoint-induced slowdowns to the single event loop turn that contains the timepoint.

4.1.2 Debugger Bookmarks

Whenever a developer wants to repeatedly inspect two or more program states using a debugger's state inspection tools, she must first suspend execution at an execution instant using timepoints, breakpoints, and/or debugger commands (Section 2.1.3). *Debugger bookmarks* are a Dolos extension for saving and quickly restoring execution instants reached via debugger commands for which the associated timepoint is not known. A debugger bookmark consists of a base timepoint (corresponding to when the debugger first pauses) and a list of debugger commands (`step-into`, `step-over`, etc.) that can be executed to reach the execution instant from the timepoint. A fresh "replay log" of debugger commands is preëmptively captured for the duration of each event loop input. To re-execute up to a debugger bookmark, Dolos seeks to the bookmark's timepoint, and then simulates commands from the debugger replay log. Debugger bookmarks make it

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

possible to initiate a dynamic analysis from any state during playback and revert back to the initial state after an analysis has complete.

4.2 Extracting Program States

A captured execution contains an enormous corpus of ephemeral runtime data that is only available as execution proceeds. This data includes both *operations*—user events, callbacks, control flow, program operations, visual output—as well as *aggregate data*, such as the state of the stack and heap, profiling information, and logged values.

The main difficulty in working with this data is that it is produced as a side-effect of execution; extracting data inherently requires re-execution. If a captured execution is being replayed non-interactively[5], then this is not a problem. In contrast, retroactive developer tools are used directly by humans, whose expectations raise interesting questions. How can tools extract program states at interactive rates? How can tools extract states without disrupting task context?

4.2.1 Scanning Algorithms[6]

As a developer's information needs change, retroactive tools should be able to collect different program states using different sets of instrumentations. To support this, Dolos provides a scanning command that automatically re-executes a contiguous region of the captured execution with instrumentation enabled. This command allows a tool to asynchronously collect past program states without reimplementing common low-level tasks, such as scheduling playback. It also allows for data collection to be virtualized in environments where background scanning is possible (see Section 4.2.2).

The scanning algorithm is illustrated by cases in Figure 4.1. Scanning has two design

[5]This dissertation focuses on interactive, developer tool-oriented uses of deterministic replay. Section 8.4 describes several batch, non-interactive use cases for deterministic replay.

[6]The scanning algorithm presented here was originally created as a CSE 503 course project, and has been improved and generalized since then.

constraints: execution should return to the initial timepoint (to restore the tool user's task context), and redundant re-execution should avoided when possible (to make scanning fast). Scanning is agnostic to the particular instrumentation approach; the scanning command delegates enabling and disabling instrumentation to the caller, invoking these operations at appropriate times during playback. In common cases (Figure 4.1(a–d)), scanning requires instrumented re-execution up to and over the specified region, and normal re-execution back to the starting timepoint. In the degenerate case (Figure 4.1(e)) where the starting timepoint is within the specified region and instrumentation cannot be enabled at the starting timepoint[7], some redundant execution is necessary to return to the starting timepoint.

As a relatively late feature addition to Dolos, scanning has only been used sparingly by the tools in this dissertation. I used scanning to implement proactive detection of breakpoint pauses (surfaced as Breakpoint Radar in Section 5.2.4) and automated collection of probe samples (Section 5.3). It was also used in experiments to gather profiling data retroactively over a user-selected region. As future work, it could be used by Scry (Chapter 6) to capture past visual states.

4.2.2 Virtualizing State Extraction

A significant limitation of extracting program states from recordings is that a tool user must idly watch as the web program is re-executed during scanning. Ideally, this data collection would be performed in a *background* rendering engine process, without controlling execution of the *foreground* web program. If extraction of program states be virtualized, then asynchronous, parallel, or distributed scanning algorithms could accelerate extraction of these states and enable a less disruptive user experience.

This section discusses prerequisites for virtualizing the extraction of program states,

[7]Breakpoints and probes can be modified any time and take effect immediately. Bytecode instrumentation of JavaScript requires recompiling all code blocks, which can only be done when the call stack is empty (i.e., at the beginning or end of an event loop turn). Source instrumentation of JavaScript happens when scripts are initially parsed, and thus requires re-execution from the start of a recording segment.

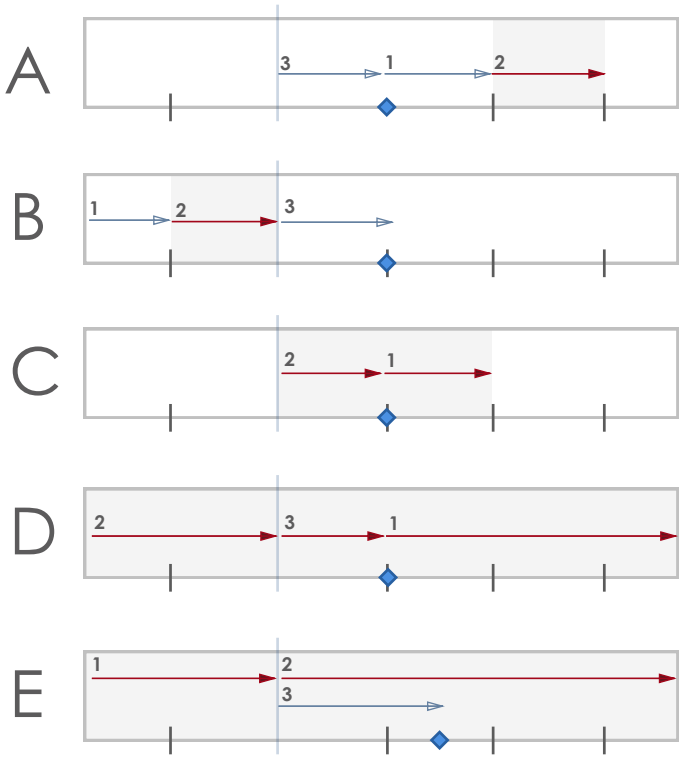

Figure 4.1: Diagrams showing different cases of the scanning algorithm. Each diagram represents a recording with 2 segments and 6 event loop turns; hash marks denote event loop turn boundaries, and vertical rules denote recording segment boundaries. Red arrows denote instrumented execution, gray arrows denote uninstrumented execution, and numbers denote sequencing of re-execution commands. The target region (shaded in gray) must always be covered by instrumented execution, and execution must return to the starting execution instant (blue diamond).

and sketches how an asynchronous "worker" architecture might work. Implementing and evaluating this design in WebKit requires substantial engineering, and is left to future work.

The requirements for splitting the extraction of program states are similar to the requirements when programming with libraries that implement task and data parallelism [98]. Specifically:

- Data collection should be *divisible* (i.e., segregated by event loop turn) so that different workers can efficiently split re-execution of the specified region.

- Collected data should be *combinable* so that data from different regions can be collated together and redundant data can be deduplicated.

- Collected data should be *serializable* so that data collected by other processes or machines can be remitted to the retroactive tool's process on the developer's machine.

Figure 4.2 shows a proposed design for virtualized extraction of program states. At an architectural level, a primary process coordinates the instrumented re-executions performed by one or more worker processes. As program states are collected by specific agents[8] during re-execution, they generate state update messages which are replicated back to the corresponding agent in the primary process. When data is collected locally in the primary process, state update messages do not need to be replicated.

4.3 Mitigating Replay Faults

Web browsers contains many sources of nondeterminism, and invariably, not all of them are handled properly by a research prototype like Dolos. Adding code to control all sources of nondeterminism is a huge engineering effort for any runtime. Rendering engines receive many code changes per day, and a nontrivial fraction of these changes introduce or modify sources of nondeterminism. In many cases—such as obsolete or experimental APIs or configurations—adding replay support is simply not that important. Thus, waiting until "everything works" is a not a good strategy for building out and adopting deterministic replay in production-quality runtimes. Instead, implementors should build deterministic replay systems incrementally and anticipate that errors or

[8]The Web Inspector [175] developer tool suite splits its functionality into multiple agents that instrument distinct parts of the rendering engine. Some agents include those for the Timeline, DOM modifications, JavaScript Debugger, Scry, Profiler, and Local Storage.

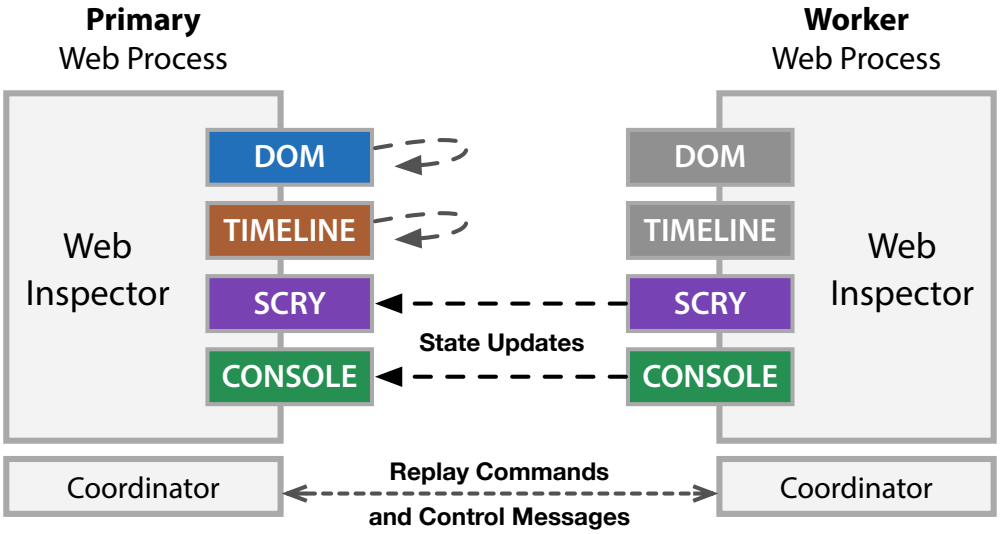

Figure 4.2: Diagram of a proposed architecture for virtualized extraction of program states.

failures may occur during capture and replay, and design mechanisms to detect these errors and safely recover or abort playback. This section explains invariants and oracles that Dolos uses to detect and mitigate a fault's corresponding errors (benign divergence) and failures (JavaScript divergence). For consistency, in this section I refer to a lack of control over a specific source of nondeterminism as a fault/defect within Dolos, following the definitions in Section 1.3.

Dolos considers a replay error to have occurred when an execution differs in any respect that could *potentially* cause JavaScript code to execute in a divergent way. All errors represent a defect in Dolos' handling of nondeterminism. However, most errors are "harmless": JavaScript still executes the same way because the nondeterminism was inconsequental to the execution (i.e., a user input event fired DOM events for which there were no event listeners). These occurrences of "benevolent nondeterminism" can be safely ignored by a tool user, but are very useful clues to a tool developer when isolating defects in the replay infrastructure. To match these expectations, the error detection techniques in this section are only performed in debug builds of Dolos. In normal builds,

Dolos attempts to continue playback as long as possible, stopping only when JavaScript code requests invalid memoized inputs from the replay log.

Dolos detects some replay errors by capturing additional bookkeeping during capturing and replaying. Dolos uses a variety of deterministic metrics that can be cheaply sampled, such as DOM node counts, DOM event dispatch counts, and API call counts. Dolos samples these metrics before every event loop turn and looks for discrepancies between the data collected during capturing and replaying. In practice, collecting metrics once per event loop turn is cheap and provides sufficient detail to quickly isolate errors to the preceding event loop inputs. In the past few years, metrics comparisons have helped me find and address obscure sources of nondeterminism, such as unintentional interactions with the resource cache, asynchrony in the HTML parser, initial window dimensions, default form element focus state, and improper delivery of event loop inputs to nested `<iframe>` elements.

Dolos detects other replay errors by checking critical invariants whenever JavaScript code begins to execute, or as a simulated event loop turn finishes. Recall that any event loop input that causes JavaScript code to execute is considered nondeterministic and must be captured and replayed by Dolos. During playback, these event loop inputs are dispatched in order to initiate processing by the rendering engine. Thus, Dolos checks the invariant during playback that *whenever JavaScript code executes, there should exist an initiating event loop input that is currently being synchronously dispatched by Dolos*. If the opposite is discovered—that JavaScript is running with an unknown initiating event loop input—then this is evidence of unhandled nondeterminism. In the past two years, this invariant has detected the majority of unhandled nondeterminism. Another invariant is that *processing of an event loop input should always require the same number of memoized inputs*. If a different number memoized inputs are requested, this is evidence of more significant unhandled nondeterminism that caused JavaScript execution to diverge. Violations of this invariant are more difficult to debug because the error that caused the mismatch might have happened much earlier in the execution.

4.4 Summary

This chapter presents several extensions to the core deterministic replay infrastructure of Chapter 3 that make it suitable for use by retroactive developer tools in real-world scenarios. With these extensions, Dolos can be used to extract and navigate to program states in a captured execution. Together, the extensions in this chapter make the following contributions:

- A scheme for uniquely identifying and re-executing to any executed statement.
- An algorithm for quickly collecting program states from a captured execution using arbitrary instrumentation.
- A design for distributing the collection of past program states to worker processes.
- Invariants and oracles that can detect replay errors and failures.

Chapter 5

INTERFACES FOR NAVIGATING CAPTURED EXECUTIONS[1]

The techniques for collecting past program states described in previous chapters dramatically change the problems that tool creators face: rather than runtime data being scarce and difficult to collect, runtime data is now abundant and requires the use of visualization and retrieval techniques to avoid information overload. This chapter investigates interfaces that expose these new capabilities and runtime states in ways that enable a developer to more easily answer her own questions about program behavior.

The techniques of previous chapters make it possible to automatically reproduce behaviors, but reproducing a behavior is just one sub-task in a complex program understanding task. To make further progress, a developer uses tools to satisfy her information needs (Section 2.3.2) and answer her questions about program behaviors. Since runtime states are abundant, retroactive developer tools are primarily concerned with *finding* relevant program states, *visualizing* program states over time, and *navigating* between relevant program states.

This chapter describes two retroactive developer tools that exemplify complimentary approaches for *navigating* through a captured execution[2]. Timelapse (Section 5.2) is a timeline-oriented visualization that allows a developer to navigate a recording via its input events. Data probes (Section 5.3) are an alternative interface that allows a developer to retroactively log program states and navigate via these ad-hoc logged states. The chapter begins by presenting design issues applicable to any replay interface, and then introduces each design and its implications using running examples.

---

[1]The results in this chapter appear in part in Burg et al. [30] and Burg et al. [31]. Some features were upstreamed to WebKit and shipped in Safari 8; see Section A.1.5 for details.

[2]I particularly focus on *navigation* as a lens for understanding changes in a developer's program un-

5.1 Controls for Capture and Playback

A user of a retroactive tool should always know whether the web program they are looking at is being captured, replayed, or suspended, and why. The replay system's state is significant because it determines what other operations and capabilities are available. For example, Dolos ignores "live" user inputs during replaying; if there were no visual indication that playback is in progress, then the browser may simply appear to be buggy or stuck because it seemingly ignores all user input. To surface system state, this chapter's prototypes use a combination of page overlays (such as dimming the inspected page or adding a colored border), enabling or disabling commands (visualized through the buttons described in this section), and visualization-specific aids such as animations.

Another usability issue is the matter of how to communicate replay errors and failures (Section 4.3), and how to proceed if an error is recoverable. Timelapse logs a warning to the console when known-unsupported nondeterministic APIs are used. If a replay failure occurs (i.e., Dolos detects divergence of JavaScript execution), a dialog appears that allows the user to abort playback, continue using best-effort playback, or file a bug using the created recording as a failing test case.

A tool for revisiting past program states must expose controls for starting and stopping the capture or playback of an execution. Like prior work [109, 170], both of the retroactive developer tools presented in this chapter expose VCR-like buttons (seen at the bottom of Figure 5.1) that control the basic operation of the replay infrastructure. Below, I enumerate each button and its purpose.

- The "record" button (Figure 5.1(6)) starts and stops recording. When initially pressed, Dolos will restart (i.e., reload) the current web program and begin capturing its execution. During capturing, a developer can interact with the web program to re-

derstanding processes. Recent applications of information foraging theory (IFT) to program understanding [57, 95] provide a deeper examination of navigation as one of the key observable actions during development tasks. Despite the obvious connections to IFT, the user interface designs in this chapter were not explicitly motivated by recent IFT work; this would be an interesting avenue for future work.

produce behaviors of interest. Pressing the record button again will finalize the recording and suspend execution immediately.

- The "play/pause" button (Figure 5.1(7)) initiates or suspends playback depending on the current replay system state. By default, the play button will resume or restart playback at $1\times$ speed to simulate the timings captured in the original execution. The pause button will halt further dispatch of event loop inputs by Dolos, effectively starving the web program of further events. When paused or during playback, the web program does not accept live inputs.

- The "lock/unlock" button (not pictured[3]) indicates whether or not live user inputs are being processed by the rendering engine. During capturing, the button is always unlocked, indicating live inputs are being captured; during playback, the button is usually locked to indicate that live inputs are being ignored. Pressing the locked button disables Dolos' input blocking functionality, allowing a user to intentionally diverge execution during playback by feeding it new user input events.

- The "eject" button (not pictured) allows a developer to discontinue capturing or playback with the currently-loaded recording. As the replay infrastucture can only load one recording at a time, this disc player metaphor clearly communicates the state of the replay system.

To get a sense for how these basic replay controls outlined above might be used in real development tasks, we conducted a small pilot study with our first prototype interface (Figure A.1, p. 132). 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, several

[3]The lock button was removed from later prototypes. In user studies, users did not understand its purpose. Later prototypes adopted the "eject" button, which implicitly performs an unlock if clicked while locked.

common use cases emerged: "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.

5.2 Navigating Executions via Input Events

The basic replay controls outlined above allow a developer to capture an execution and replay linearly, but in many situations, a developer wants to seek *non-linearly* through an execution to specific behaviors or interactions. Timelapse is a retroactive developer tool that provides a time series visualization of a captured execution. Timelapse uses discrete timelines to visualize classes of event loop inputs over time, and a linked data table to reveal low-level details about each input event. A developer can use these interactive visualizations of program inputs to replay up to interesting points in the recording, and then use the debugger or other tools to further understand program behavior. The remainder of this section explains Timelapse's visual encodings, interface design, and interactivity, and then walks through Timelapse's functionality using a running example.

5.2.1 Interface Design

Timelapse visualizes a captured execution's nondeterministic inputs over time using multiple timeline widgets (Figure 5.1). Timelines are appropriate for visualizing executions because they can compactly summarize time series data, such as past program states and inputs. Timelines of execution have been a core feature of the Web Inspector [175] for a long time. Compared to these existing timelines which visualize traces of execution events, Timelapse's timelines support navigation to past execution instants in the recording by interacting with data points in the visualization. As the first interface built atop of Dolos, Timelapse also served a dual purpose during development as a window into the inner workings of Dolos, showing detailed information about each event loop input and other diagnostics, which are not discussed further.

Figure 5.1: 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).

Timelapse's timelines use a simple visual encoding to summarize inputs over time on a single axis. Each timeline represents a category of event loop inputs, such as network, asynchronous work, and user events; each timeline maintains its own color scheme. The x-axis is used as a linear time scale; each input is plotted according to its timestamp. Circles are used to encode the density of inputs per unit time; circle radius is determined using static buckets. Slight variations in transparency are also used as a redundant encoding of input density. Single circles can be clicked to see their constituent inputs, or double-clicked to seek execution to the first input within that circle.

In addition to heatmap-like timelines, Timelapse uses a linked overview and details table to support zooming, filtering, and details on demand. During playback, a red cursor in the timelines and details table surfaces the current progress of execution. The red cursor can also be dragged and dropped in the overview, timelines, or details table to seek to specific inputs. The rest of this section motivates Timelapse's user interface elements and interactions by example.

5.2.2 Example: (Buggy) Space Invaders

Steph, a new hire at a fictitious game company, has been asked to fix a bug in a JavaScript version of the Space Invaders video game[4]. 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, helper libraries, and responds to user input. 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 5.2).

Steph is unfamiliar with the Space Invaders implementation, so her first step towards understanding and fixing the multiple-bullet bug is to figure out how to reliably reproduce it. This is difficult because the failure only occurs in specific game states and is

[4]Space Invaders: `http://matthaynes.net/playground/javascript/glow/spaceinvaders/`

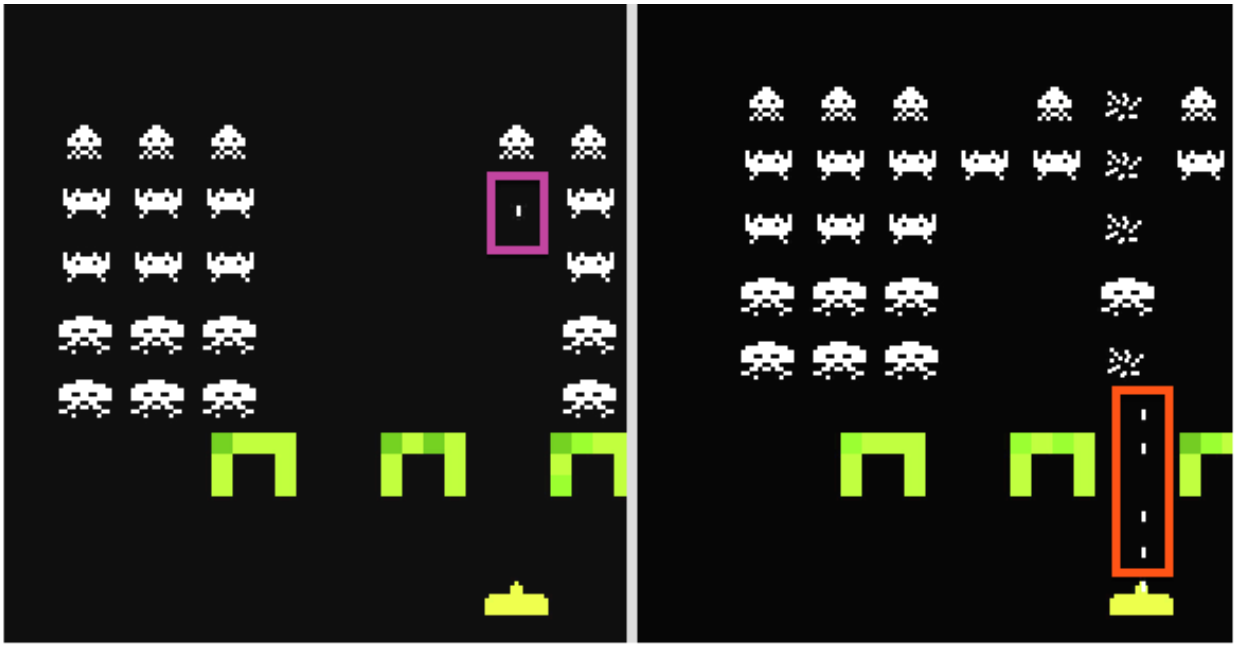

Figure 5.2: Screenshots of normal and buggy Space Invader game mechanics. Only one bullet should be in play at a time (shown on left). Due to misuse of library code, each bullet fires two asynchronous `bulletDied` events instead of one event when it is destroyed. The double dispatch sometimes enables 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.

influenced by execution conditions outside of her control, such as random numbers, the current time, or asynchronous network requests. To reproduce the bug without Timelapse, Steph would have to multitask between playing the video game and writing down reproduction steps. Once she establishes reliable reproduction steps, she could then use breakpoints to further understand the bug. But, breakpoints might themselves affect timing, making the failure harder to trigger or requiring modified reproduction steps.

Using Timelapse, Steph begins capturing program behaviors by pressing the "record" button (Figure 5.1(6)). Then, she plays the game until the failure manifests, and then she presses the button again to stop capturing. Once capturing stops, a timeline of the entire execution appears.

To find code that's relevant to the failure, Steph needs to know which specific user input—and thus which event handler invocations—caused the second bullet to appear. After she finishes creating the recording, an overview of the captured execution is displayed (Figure 5.1(2)). First, Steph plays back the recorded execution to confirm that it was captured correctly and to find the relevant section of the recording. Immediately after she sees the failure manifest, she pauses playback. She then adjusts the zoom interval (Figure 5.1(3)) to contain the last few keystrokes prior the paused time, and filters out non-keyboard inputs. With only a few candidate inputs, she advances playback a single keyboard input event at a time 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, and can use other tools (such as logging or breakpoints) to debug the event handler.

Without Timelapse, it would not be possible for Steph to isolate the failure to a specific keystroke in this way. She would have to simultaneously enter keystrokes and use the debugger; the latter would mask the failure because the timing of keystrokes is critical to reproducing the failure. Instead, she would have to insert logging statements, repeatedly reproduce the failure to generate logging output, and scrutinize logged values for clues leading towards the root cause.

5.2.3 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. Using a Dolos extension called *debugger bookmarks* (Section 4.1.2), Steph saves several bookmarks 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 state reached via the debugger. Debugger bookmarks support the concept of temporal focus points [76, 149], which are useful when a developer wants to revisit information—such as the program's state at a breakpoint hit—that is only available

at certain points of execution.

With traditional tools such as a breakpoint debugger, 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 program states or the effects of different commands.

5.2.4 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. *Breakpoint radar* is a Timelapse feature (built using the data scanning algorithm described in Section 4.2) that automatically scans potential breakpoint hits over the entire execution. Detected breakpoint hits are visualized on a separate timeline (Figure 5.3(3a)). Steph can easily see which keystrokes created bullets and which did not. 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.

Using a breakpoint debugger, 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.

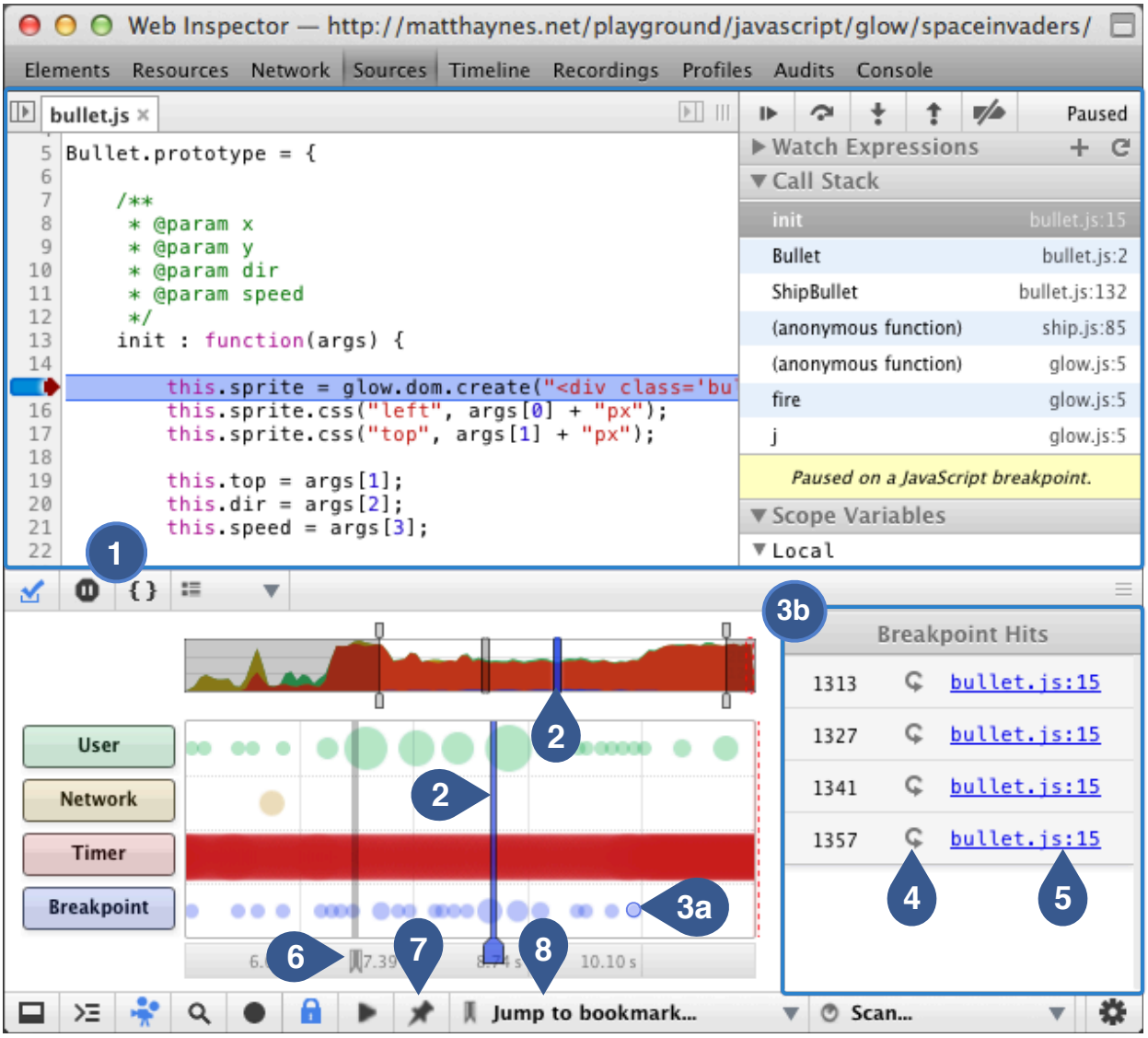

Figure 5.3: Timelapse's visualization of debugger status and breakpoint history, juxtaposed with the existing Sources panel and debugger (1). A blue cursor (2) indicates that replay execution is paused at a breakpoint, instead of between user inputs (as shown in Figure 5.1). Blue circles mark the location 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).

5.2.5 *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[5], 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).

Timelapse provides an interface for capturing and replaying input events that determine an execution, eliminating the need for Steph to repeatedly reproduce the Space Invaders failure. Timelapse allows Steph to interactively navigate within the captured recording via its event loop input events. Steph can revisit past program states by first replaying to the proceeding event loop input event, and then using other developer tools such as breakpoints. Chapter 7 presents an exploratory user study to discover the benefits

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

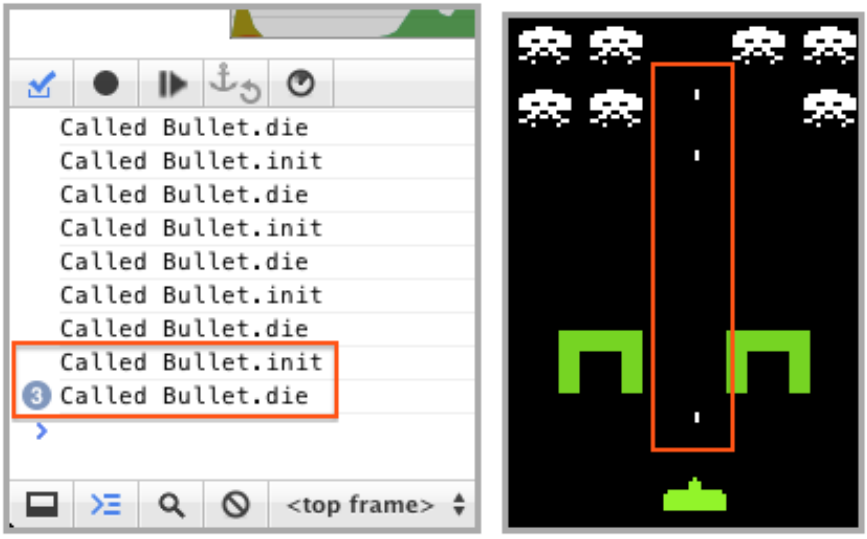

Figure 5.4: Screenshots of the logging output window and the defect's manifestation in the game. Steph added logging statements to the `die` and `init` methods. At left, the logging output shows the order of method entries, with the blue circle summarizing 3 identical logging outputs. According to the 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`.

and barriers of the Timelapse interface as described in this section.

5.3 Navigating Executions via Logged Runtime States

The Timelapse interface supports non-linear navigation within a captured execution, but navigating to specific program states is still cumbersome. To suspend execution at a specific program point, a developer must isolate and replay to the preceding input, and then switch to using a breakpoint debugger to drive execution to a specific statement.

According to several studies [83, 141] (and common sense), it is easier for a developer to navigate an execution via its *outputs*, rather than by its inputs. A program's outputs are often closely related to important program states, and developers often work backwards from outputs when attempting to understand runtime behavior [82]. To this end, this

section develops a second retroactive tool designed around the idea of output-oriented navigation. In contrast to Scry (Chapter 6)—which supports navigating through an execution via visual outputs—this section supports navigating through an execution via textual outputs produced by logging program states.

When investigating a program's outputs, a developer may wish to jump to the instant when a logged output was created. *Time-indexed outputs* are a user interface feature designed to provide this capability. Time-indexed outputs enable a developer to see the logged output they want to investigate and, with a single click, jump to the exact moment in the captured execution when a program statement that produced the output. 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. Behind the scenes, each time-indexed output is tagged with a Dolos timepoint (Section 4.1.1) which uniquely identifies the statement execution that produced a specific output.

During debugging tasks, a developer may spend a significant fraction of their time inserting logging statements in source code in order to inspect runtime state at specific program points. *Data probes* allow a developer to retroactively add logging statements to a captured execution without editing program text and without re-executing the program. Using data probes, a developer can interactively discover, compare, and navigate to interesting program states in the past or the future without excessive planning or manual effort. 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*. 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. Data probes and probe samples are saved across multiple playbacks of a captured execution. Probes are currently implemented as a special breakpoint action that evaluates the probe's expressions.

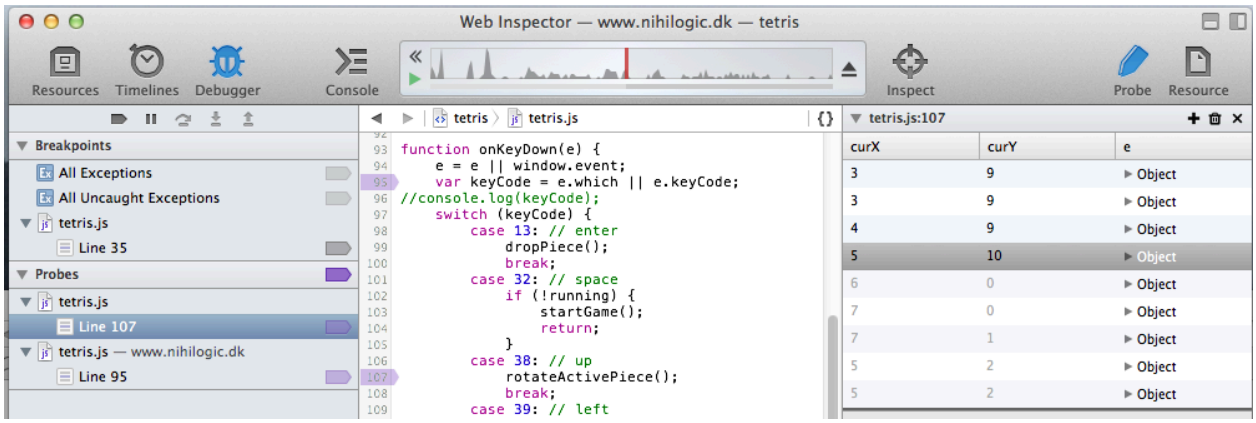

Figure 5.5: The probes interface while debugging DOMTris.

5.3.1 Interface

The probes user interface (Figure 5.5) supports comparing, relating, and navigating to probe samples. The probes sidebar (Figure 5.7(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 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.

Time-indexed outputs do not require a new user interface. Whenever a developer wishes to replay back to a time-indexed output, she simply double-clicks on the relevant logged output in the console or other windows that display console output. rather, existing mechanisms for logging runtime states now additionally tag logged outputs with timepoints.

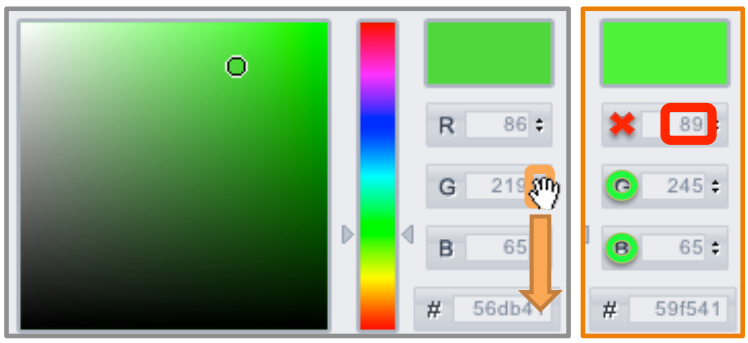

Figure 5.6: The Color Picker widget. When the G color component is adjusted downward (left), the R component unexpectedly changes (right).

5.3.2 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[6] is a jQuery plugin that implements a color picker widget for RGB (red, green, blue) and HSV (hue, saturation, valence) color spaces. Karla, a fictional developer, uses the widget in her web program. She is investigating a bug that manifests itself when a user manipulates the color picker's color component sliders (Figure 5.6). Each slider should adjust the value of one red, green, or blue (RGB) component independently of the 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.

[6]Colorpicker: `http://www.eyecon.ro/colorpicker/`

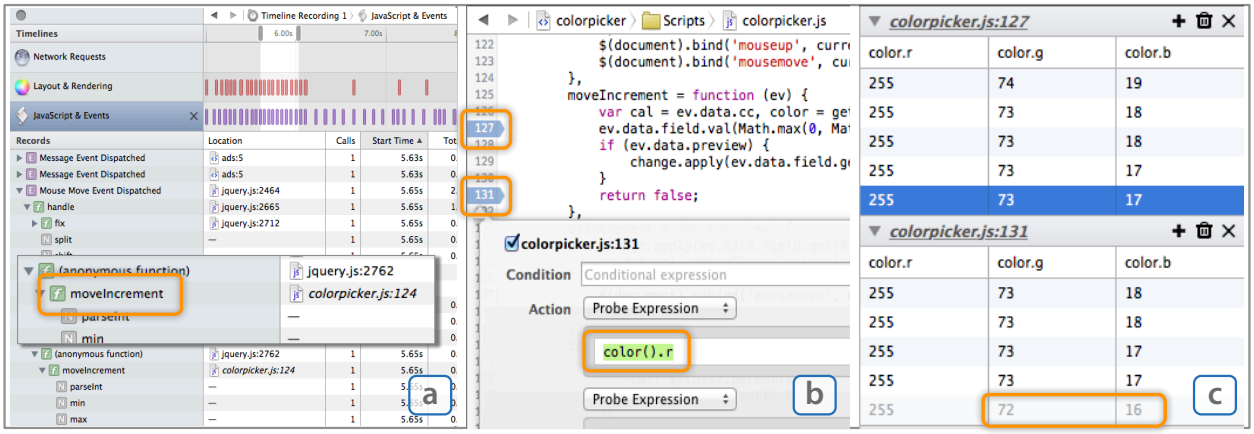

Figure 5.7: Using data probes to revert execution to relevant program states. In (a), Karla inspects the captured recording to find the `moveIncrement` drag event handler (highlighted). In (b), she adds two *data probes* (at lines 127 and 131) to log how color component values changed. In (c), she reverts execution directly to a probe sample (top, selected row) that immediately precedes erroneous component values (below, highlighted).

Karla starts by using a record/replay tool (such as Dolos) 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 program, 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

```
[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 5.8: 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.

to the console. After using the built-in timelines view (Figure 5.7(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 5.7(b)). These data probes capture the value of the expressions `color.r`, `color.g`, and `color.b` (Figure 5.7(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 5.7(c)) begins to populate with probe samples. Looking at temporally-ordered probe samples in the console (Figure 5.8), Karla quickly sees a few 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 5.7(c) and Figure 5.8). 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.

5.4 Summary

This chapter presents two retroactive developer tools that support different navigation patterns by using the replay and state extraction capabilities of Dolos. Timelapse is a timeline-oriented visualization that enables a developer to navigate a captured execution via its event loop inputs. Data probes enable a developer to retroactively log ad-hoc program states and navigate directly to program states. These prototypes make the following contributions:

- 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.
- Examples of tools that use timepoints, scanning, and other abstractions introduced in Chapter 4.

Chapter 6

EXPLAINING VISUAL CHANGES IN WEB INTERFACES[1]

The previous chapter presented two retroactive tools that enable a developer to navigate a captured execution via its inputs or via logged program states. However, both of these methods of navigation require familiarity with the web program's source code to be truly useful. If a developer is inspecting unfamiliar code, then they still must add multiple probes to try and find relevant source code. Before they can use tools like breakpoints and debugger bookmarks, a developer must spend considerable effort to find relevant code.

This chapter develops an alternative mode of navigating captured executions that requires less up-front knowledge of relevant source code. Instead of requiring a developer to identify important inputs or logged states, Scry—the tool developed in this chapter—allows a developer to directly select and compare specific visual output examples. Once a developer has identified state differences, she can look at a list of execution instants that are causally related to the state difference. In this way, the retroactive tool of this chapter augments other retroactive tools in this dissertation by providing a powerful means for finding relevant program states and excution instants.

6.1 Motivation

Web developers increasingly look to existing designs for inspiration [53], to learn about new practices or APIs [25], and to copy and adapt interactive behaviors for their own purposes [155]. Web programs are particularly conducive to reuse because web pages are widely available, distributed in source form, and inspectable using tools built into web

[1]The results in this chapter appear in part in Burg et al. [32].

browsers.

Unfortunately, when a developer finds an interactive behavior within a third-party web program 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 is a challenging process [86]. 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 source of the client-side web program 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 cascading style sheets (CSS) imperative JavaScript, opaque Document Object Model (DOM) API, and notoriously complex layout/rendering algorithms makes the feature location task prohibitively difficult. Moreover, as web programs 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 [123], 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 [5, 30, 109], 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 [30, 95]. In other programming environments, research prototypes with this functionality all incur run-time overheads that limit their use to post-mortem debugging [71, 82, 120].

To address these gaps, I developed 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. 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. 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. 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 chapter 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 by presenting several real-world case studies and limitations of the approach.

6.2 Example: Understanding a Mosaic Widget

To illustrate Scry in use, consider Steph, a fictitious contract web developer who is overhauling a non-profit organization's web presence to be more engaging and interactive.

Figure 6.1: A picture mosaic widget[2] 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.

While browsing another page[2], she finds a compelling picture mosaic widget (Figure 6.1) 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, Steph 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 Steph 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. Steph 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

[2] The mosaic widget was found on the following page: `https://www.mozilla.org/en-US/mission/`

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 Steph is unfamiliar with the example's code.

Using Scry, Steph 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, Steph's workflow is grounded by output examples, captured DOM and CSS states, and specific lines of JavaScript code. Steph starts by finding the mosaic's corresponding DOM element using the Web Inspector, and then tells Scry to track changes to the element (Figure 6.2(a)). As mosaic tiles update, Scry captures snapshots of the mosaic's internal state and visual output. After several tile transitions, Steph stops tracking and browses the collected snapshots to identify visual states before, during, and after a single tile changes images (Figure 6.2(b)).

Steph 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 6.2(b)). For each selected screenshot, Scry shows the small subset of the page's DOM (Figure 6.2(b)) and CSS (Figure 6.2(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, Steph 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 6.2(c)). Seeing that the tiles' `background-image` and `opacity` properties have changed, Steph now knows which CSS and DOM properties the mosaic uses to implement the cross-fade.

To find the code that causes these changes, Steph 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

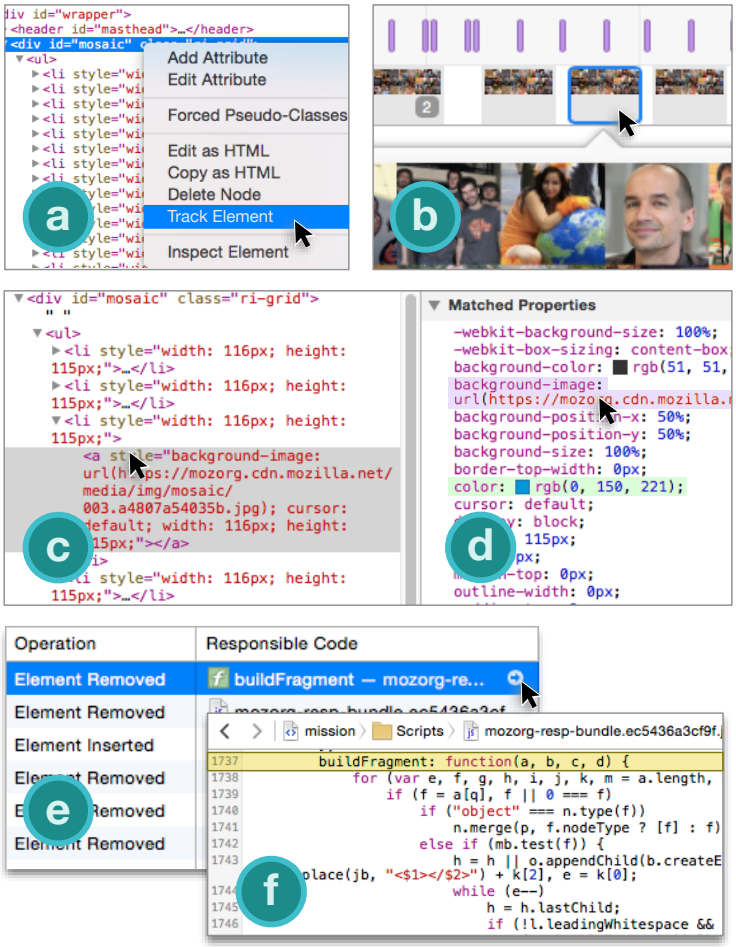

Figure 6.2: An overview of the Scry workflow for localizing visual changes. Steph first uses the Web Inspector to go from the mosaic's visual output to its DOM elements. Then, she uses Scry to track changes to the mosaic element (a), select different visual states to inspect (b), and see the DOM tree (c) and CSS styles (d) that produced each visual state. To jump to the code that implements interactive behaviors, Steph uses Scry to compare two states and then selects a single style property difference (d). Scry shows the mutation operations responsible for causing the property difference (e), and Steph can jump to JavaScript code (f) that performed each mutation operation.

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 6.2(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 6.2(e,f)). Steph now knows exactly how the widget's JavaScript, DOM, and CSS code works together to animate a tile's cross-fade transition. If Steph wants to modify the animation code, she now knows several places within the code from which to expand her understanding of the program.

6.3 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 [144]. 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.

6.3.1 Design Rationale

I designed Scry to directly address the information overload a developer encounters when performing feature location tasks [144]. 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. I explain each of these points below.

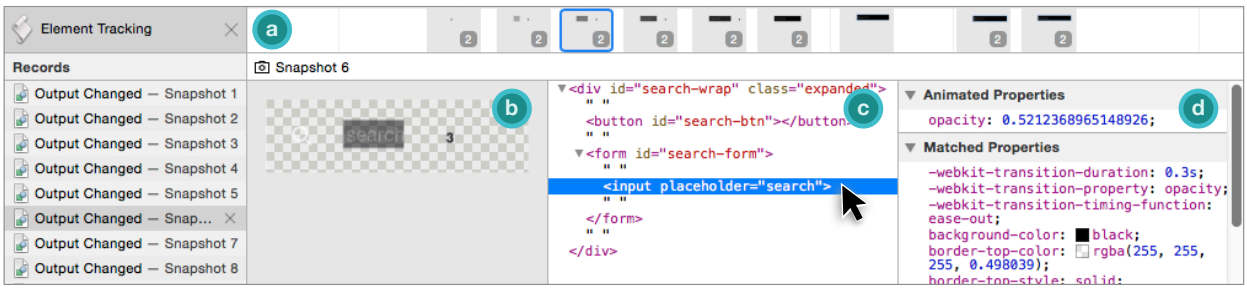

Figure 6.3: Scry's timeline interface. Multiple screenshots of a tracked element are shown in the timeline strip (a). When the developer selects an output example from the timeline, Scry shows three views for it: (b) the element's visual appearance, (c) its corresponding DOM tree, and (d) computed style properties for the selected DOM node.

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 6.3). 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, a developer can do so without considering scripted behaviors and other interactions. While most interactive behaviors are scripted with JavaScript[3], 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. To support this task, Scry juxtaposes each screenshot with its corresponding DOM tree and computed CSS style properties (Figure 6.3).

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 6.4). 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. To prompt a user to test their mental model against a small, understandable example, Scry juxtaposes small changes in internal state with the corresponding visual outputs. These differences 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 *mu-*

[3]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.

tation 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 [95] to find other relevant code upstream from the mutation operation. Initially, the user is presented with a list of recorded operations for one state difference; the user can browse these operations to understand how the state changed. Once the user wants to see the source code responsible for these mutation operations, they click on a specific operation to see where it was performed. Many mutation operations originate from source code (Figure 6.5), 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.

6.3.2 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* ofinterest, using existing tools such as an element inspector or DOM tree viewer. Once the developer issues Scry's "Start Tracking" command (Figure 6.2(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. (I later explain how these tracking capabilities are implemented.)

The element tracking timeline (Figure 6.3(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

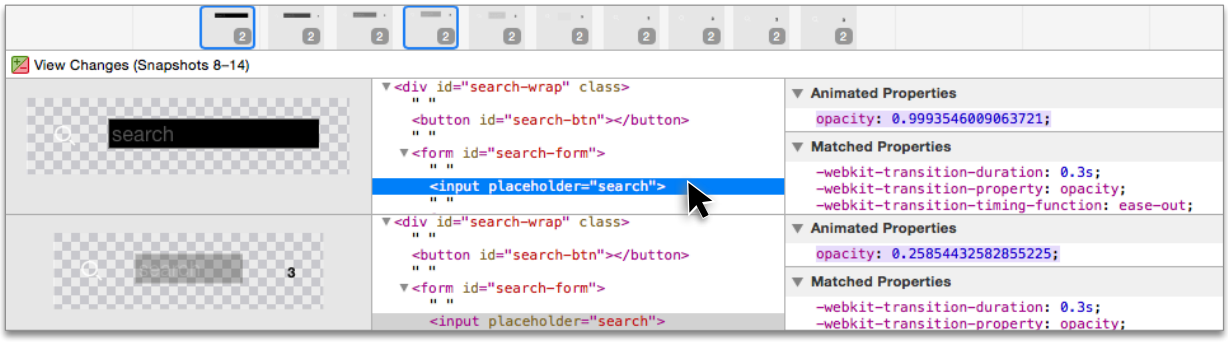

Figure 6.4: Scry's interface for comparing visual state snapshots. Two screenshots are selected in the timeline, and their corresponding CSS styles and DOM trees appear below. Differences are highlighted using inline annotations; here, the `opacity` style property has changed. Additions, removals, and changes are highlighted in green, red, and purple, respectively.

interactions or event types.

6.3.3 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[4]. After a developer has captured relevant output states of the target element, she then selects a single screenshot from the timeline (Figure 6.3(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 6.3). 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 6.3(c)), Scry shows the element's matched CSS styles (Figure 6.3(d)).

[4]Scry does not directly explain causal relationships between inputs and outputs in the style of Whyline [82]. 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.

6.3.4 *Comparing Internal States*

Using Scry's comparison interface (Figure 6.4), 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 6.4) 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.

6.3.5 *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 (Figure 6.4(a)). Then, Scry changes views to show the difference alongside a list of mutation operations (Figure 6.5(b)) that caused the difference. Each operation includes a JavaScript stack trace that shows the calling context for the mutation

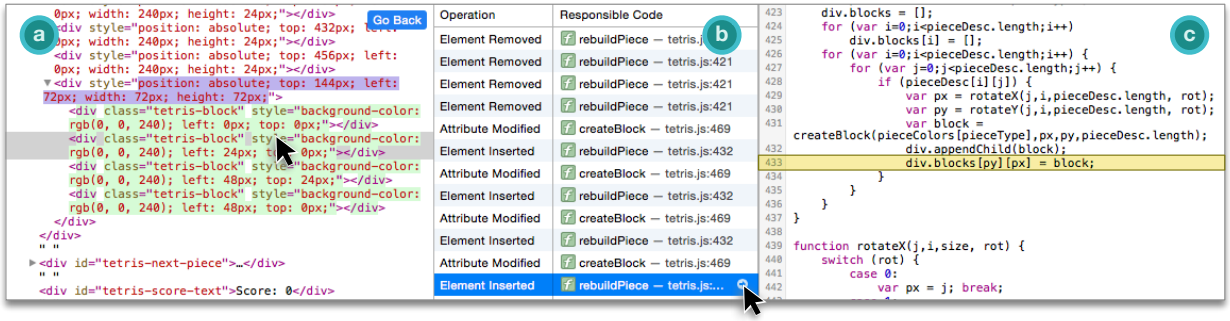

Figure 6.5: Scry's interface for showing the operations that were responsible for a change. The left pane (a) shows a single difference being investigated (the second added `<div>`). The center pane (b) shows a list of mutation operations that caused the change. When an operation is selected, Scry shows a source code preview (c) and call stack (not shown) for the operation.

operation (Figure 6.5(c)). Using this link, a developer can find pieces of code related to a single visually-significant difference.

6.4 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.

6.4.1 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 I discuss the data a state snapshot contains and how it is captured, I first discuss how Scry decides when to capture a snapshot of a distinct visual state.

Scry differs from prior work [123] 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[5] 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

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

version is used to detect visual changes, while the background-included version is shown to the user.

6.4.2 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 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 experiences collecting view states for deterministic replay purposes (Section 3.3.3) has demonstrated that these view state inputs can be easily and

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

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.

6.4.3 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, I refer to the two snapshots being compared as the *pre-state* and *post-state*.

DOM Trees

Scry compares DOM subtrees and computes change summaries on a per-node basis. To compute an element's change summary, Scry first finds the element in both snapshots. To do this, Scry associates a unique, stable identifier with each DOM node at run time to make it possible to find the same node in two snapshots via a hash table lookup. If Scry finds the corresponding nodes in both snapshots it summarizes differences in their parent-sibling relationships, attributes, and computed styles. A node that appears in only one snapshot is reported as added or removed, and a node whose parent changed or whose order among siblings changed is reported as moved. This strategy identifies many small, localized changes (Table 6.2) that are straightforward to explain in terms of low-level mutation operations (Table 6.1). 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 [86] or tree edit distance algorithms [18]. I found these to be unsuitable for linking small state changes back to JavaScript code. Tree matching algorithms compute per-node similarity metrics, but do not try to attribute per-node dissimilarities to mutation operations. Edit distance algorithms do not directly produce per-node change summaries, and describe mutations using a minimal sequence of abstract tree operations. Web pages' mutation operations do not correspond to tree edit script operations: real edit sequences are often not minimal (for example, repeated mutations of a node's `class` attribute should not be coalesced) and include redundant but useful operations (such as replacing a subtree by assigning to `Node.innerHTML`).

Computed Styles

To compute differences between a single node's computed styles in the pre-state and post-state, Scry uses set operations on CSS property names. To determine which properties

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 6.2: 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.

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.

6.4.4 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, I first discuss the mutation operations that Scry captures as raw material for producing causal chains. Then, I 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 6.1. 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 6.1) closely mirror the most commonly used DOM and CSS APIs. These operations can be used to explain changes to DOM state, and changes to computed style properties that originate from style rules (whose rules match and un-match 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 code.

Finding Direct Operations for DOM Changes

For a specific state change (Table 6.2), 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 an `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 un-match 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[6].

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

[6]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.

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.

6.4.5 Instantiation

Scry is implemented as a set of modifications to the WebKit browser engine [174] and its Web Inspector developer tool suite [175]. Further details about the Scry prototype are available in Section A.1.6. 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.

6.5 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.

6.5.1 Expanding Search Bar

A National Geographic web article[7] contains a navigation bar with an expanding search field. When the user clicks on the magnifying glass icon, a text field appears and grows to a reasonable size for entering search terms. Without Scry, this behavior is difficult to

investigate because the animation lasts less than a second, and intermediate animation states are not displayed or persisted.

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

6.5.2 A Tetris Clone

A Tetris-like game[8] 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

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

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 [56] expose an immediate-mode API called the *virtual DOM*. Client JavaScript implements `draw()` methods that fully recreate a widget's DOM tree and CSS styles using the virtual DOM. Behind the scenes, React synchronizes the virtual and real DOM using a fast tree edit algorithm, reusing the same elements to produce visual output for unrelated model objects that happen to use the same HTML tag names. A similar problem arises with frameworks that re-render a component by filling in an HTML string template and overwriting the component's prior DOM states by setting the target element's `innerHTML` property. In this case, Scry shows the target element's entire subtree as being fully removed and re-added. Scry doesn't try to match similar nodes, but could be extended to fall back to using more relaxed similarity-based metrics [86] instead of strict identity when re-finding DOM elements.

6.5.3 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[9] 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.

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

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

We compared two snapshots to find the source of changes to the `transform` properties, and were always led back to the same line of JavaScript code. Looking upstream in the stack trace, it appeared that a JavaScript library interprets the single scroll position change and imperatively updates the `transform` style property for dozens of elements. We could not discover (using Scry) where the animation configurations for each element were specified.

From this example, we learned that Scry is of limited use for localizing code when the endpoints of JavaScript—where it directly interfaces with rendering inputs—are not easily distinguishable. Such *megamorphic* callsites to browser APIs are common when a web page calls DOM APIs indirectly through utility libraries. A simple solution would be to automatically disambiguate very active callsites based on their calling context, or allow the user to hide library code (known as "script blackboxing" in some browsers). However, for this example, simply filtering the stack traces would not lead a user to the configuration data for a parallax animation. A better solution would be to extend Scry's capabilities to include tracking of control and data dependencies through JavaScript [148]. This would require a very different technical approach, since Scry instruments native browser APIs rather than JavaScript code.

6.6 Limitations and Future Directions

Scry is a first step towards demystifying the complex, hidden interactions between the DOM, CSS layout, JavaScript code, and visual output. The capabilities that this chapter describes validate the interface concept; other explanatory capabilities could be added without significantly altering Scry's staged, example-oriented workflow. In particular, I 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.

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 [19]. Concretely, Scry could delete individual style properties from a snapshot if the resulting visual output does not differ [180]. Even with a minimal set of inputs, a more involved "white-box" approach to explaining layout [116] 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 [81, 82, 116] about how inputs are used or how outputs are derived as they funnel through the increasingly complex layout algorithms of modern web browsers.

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 [148] and event modeling [1] to extend its explanations to show an uninterrupted causal chain [82] 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.

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 programs [30] to collect snapshots and trace data from earlier instants of the execution. Prior work has demonstrated the feasibility of such an "offline dynamic analysis" [38, 39, 148], but none has integrated this technique into a user interface or web browser.

6.7 Summary

Scry demonstrates a new output example-driven method for identifying relevant source code and program states. These elements are quite difficult to identify with traditional developer tools, but are straightforward to locate using Scry. This speaks to the power of task-specific retroactive tools: not only are they more efficient to use for specific tasks,

they open new possibilities for using other retroactive developer tools that operate upon lines of source code, such as probes and time-indexed outputs.

Scry makes several important contributions to the state of the art:

- Algorithms for efficiently detecting (Section 6.4.1), serializing (Section 6.4.2), and comparing (Section 6.4.3) visual states over time.
- An algorithm for establishing causality between visual changes, state changes, and JavaScript code (Section 6.4.4).
- An interface for feature location based on comparing output and state changes (Section 6.3.1).

Chapter 7

HOW DEVELOPERS USE TIMELAPSE[1]

Prior work [52, 109, 145] asserts the usefulness of deterministic record and replay for debugging, but few deterministic replay tools have user interfaces, and none of them have ever been evaluated with real uses. To test these assumptions and discover missing infrastructure and developer tool features, I decided to put retroactive tools into the hands of professional web developers. The cold reality of having real users—if just for a few hours at a time—generated a large amount of concrete feedback about the user interface ideas embodied in the Timelapse prototype. I also found out the hard way just how difficult it is to support all sources of nondeterminism used by modern web programs.

This chapter presents a small, formative user study that investigates when, how, and for whom record/replay tools are beneficial. A secondary purpose was to demonstrate that Dolos was robust enough that external developers could pick up the tool and actually use it without encountering performance or usability problems. This initial study just covers Timelapse (Section 5.2); additional studies for Dolos, Scry, or probes have not been performed at this time, but would be interesting future work.

The study that involved Timelapse had two specific research questions:

RQ 1 How does Timelapse affect the way that developers reproduce behavior?

RQ 2 How do successful and unsuccessful developers use Timelapse differently?

7.1 Study Design

I recruited 14 web developers to participate in the study, 2 of which were used in pilot studies to refine the study design. Each participant performed two tasks. The study

[1]The results in this chapter appear in part in Burg et al. [30].

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, I randomized the ordering of the two tasks, and randomized the task for 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, I chose two tasks of medium difficulty, each with several intermediate milestones. Based on the study's results, this short-duration exploratory study was still sufficiently difficult to capture the variability in debugging and programming skill among web developers.

7.2 Participants

I recruited 14 web developers in the Seattle area. Each participant had recently worked on a substantial web program. One half of the participants were developers, designers, or testers. The other half were researchers who wrote web programs in the course of their research. I did not control for participants' prior experience with the jQuery or Glow libraries, which were used by the programs being debugged.

7.3 Programs and Tasks

7.3.1 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[2]. I 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

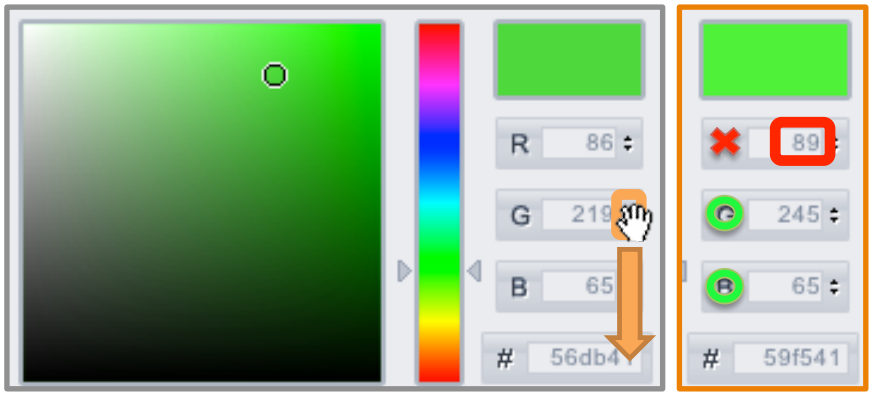

Figure 7.1: The Colorpicker widget.

web programs.

I asked participants to fix two Space Invaders defects. The first was an API mismatch that occurred when I 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.

7.3.2 Colorpicker

The other program was Colorpicker[3], an interactive widget for selecting colors in the RGB and HSV colorspaces (see in Figure 7.1). 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, value) component values or through several widgets that visualize dimensions of the HSV colorspace.

---

[2]Glow JavaScript Library: `http://www.bbc.co.uk/glow/`

[3]Colorpicker widget: `http://www.eyecon.ro/colorpicker/`

I 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 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 7.1. 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.

I 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 faults in the Space Invaders game are caused by 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 `mousemove` DOM events.

7.4 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[4]. I stopped participants when they had demonstrated successful completion to us or exceeded the time limit.

After both task periods were over, I 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. I also asked about their prior experience in bug reproduction, debugging, and testing activities. Participants who completed the study were compensated for their time.

7.5 Data Collection and Analysis

I 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. I 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.

I 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. I recorded the count and duration of all reproduction activities and whether the activity was mediated

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

by Timelapse (automatic reproduction) or unmediated (manual reproduction). Reproduction times only included time in which participants' attention was engaged on reproduction, which I determined by observing changes in window focus, mouse positioning, and interface modality.

7.6 Results

Below, I 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. I observed the opposite: because reproduction with Timelapse was so easy, participants seemed more willing to reproduce behaviors repeatedly. A possible confounding factor is that behaviors in these tasks were fairly easy to reproduce, so Timelapse only made reproduction less tedious, not less challenging. I had hoped to test whether Timelapse is more useful for fixing more challenging bugs, but were forced to reduce task difficulty so that I 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 Time-

lapse's capabilities into their debugging workflows. Corroborating the results of previous studies [124, 141], I 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. I 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.

7.7 Discussion and Summary

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

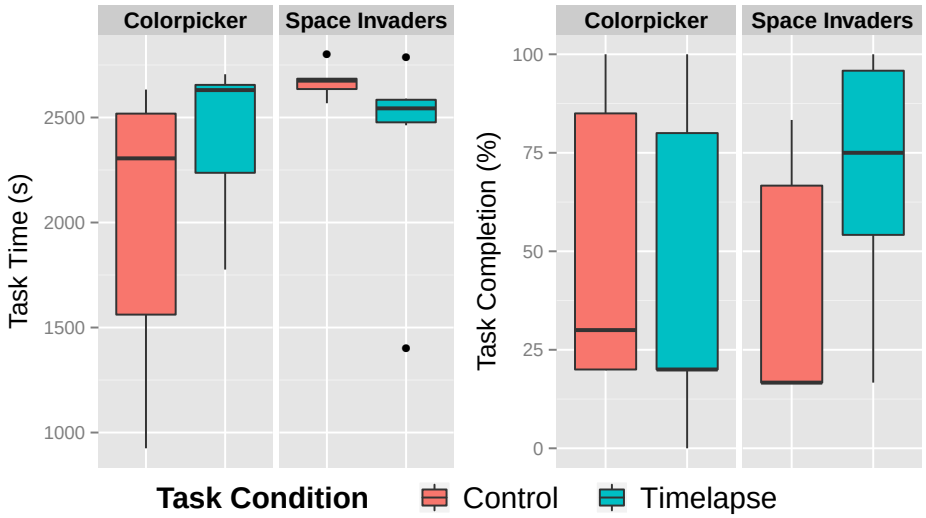

Figure 7.2: A summary of task time and success per condition and task. Box plots show outliers (points), median (thick bar), and 1st and 3rd quartiles (bottom and top of box, respectively). There was no statistically significant difference in performance between participants who used standard tools and those who had access to Timelapse.

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 reproduction 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 7.2 shows task success and task time per task and condition. In future work, I plan to study how long-term use of Timelapse during daily development affects debugging strategies. I also plan to investigate how recordings can improve bug reporting practices and communication between bug reporters and bug fixers.

Chapter 8
FUTURE WORK

Recordings produced by deterministic replay systems such as Dolos are small, but when executed, they produce vast amounts of runtime data that could be useful for many software engineering purposes. This chapter proposes new tasks and contexts which could significantly benefit from the capability to revisit past program states on demand. In particular, I explore a few applications of ubiquitous deterministic replay for software engineering: supporting collaborative bug reporting and diagnosis; synthesizing tests from recordings; supporting user-driven interactive dynamic analysis; and making it easier to empirically observe web program behavior in the large.

This chapter does not address short-term fixes and long-term architectural changes that are necessary to generalize and integrate this dissertation's contributions into existing production systems. In the short term, adopting deterministic replay is largely a matter of addressing the shortcomings of research prototypes through software engineering. Potential improvements to the reliability, features, and architecture of this dissertation's prototypes are documented in the relevant chapters, and are not further discussed here.

8.1 Collaborative Debugging

Developers often cannot fix bugs in widely deployed software because they lack the means to reproduce the bug, or in many cases, even collect rudimentary diagnostic information that may help them fix the bug. Deterministic replay can make it easier for end-users to report bugs in web programs, and for web developers to fix reported bugs using a collaborative debugging workflow. Common to both of these visions is the need to integrate and extend prior research into how recordings can anonymized and mini-

mized. Clouse and Orso [40] investigated how to anonymize sensitive, nonessential user information in field failure reports. Jin and Orso [75] investigated how to remove inessential data from field failure reports (further discussed in Section 8.2.3). Further research is necessary to integrate these techniques into deterministic replay for web programs, which have different requirements for ensuring deterministic execution.

Even when a failure can be reproduced reliably by a web developer, it is often difficult for a single developer to share their progress or seek help in diagnosing a failure. When possible, developers often prefer to synchronously interrupt a colleague to seek their help during debugging rather than post comments on an issue tracking system for later asynchronous feedback [15, 84]. I hypothesize that this tendency is partly a response to the high cost of context-switching to a debugging task. To resume a debugging task, a developer must recreate the failure, re-read and understand previous hypothesis and observations, and only then start making progress.

In addition to simplifying the process of recreating a failure, deterministic replay recordings can also enable a more documented and collaborative approach to debugging. If recordings were augmented with building blocks for collaboration such as annotations and versioning, then recordings could be the technical basis for a debugging "lab notebook". A debugging notebook would be shared amongst team members and contain the full history of a debugging session: important program states, executed commands, logged runtime values, developer notes, and associated code fixes. A notebook would contain the metadata necessary to revisit the relevant program instants mentioned in past entries. While the idea of a debugging notebook may seem far-fetched given the state of contemporary tools, this future is already a reality in the scientific computing community. IPython [125] and related tools allow users to create and share interactive notebooks that juxtapose code, results, and scientific analysis.

8.2 Creating Tests From Recordings

Using deterministic replay, it's possible to generate high-fidelity tests from a captured recording. In cases such as user interfaces, creating tests manually can be very time-consuming. Many test environments rely on text descriptions of test cases. Using deterministic replay, a user can capture their interactions with a web program, interactively permute or change inputs to see their effects on execution, and extract input sequences that should be codified as an automated test. This is a more direct way of "authoring" a test because it avoids the need to abstractly express reproduction steps using a separate vocabulary of testing commands. An interactive test editor is included with STS [147][1], which allows users to author tests with specific interleavings that would otherwise be impossible to reliably trigger. This ability would be similarly useful for authoring tests that exercise interactive behaviors in web programs.

Tests derived from replayable recordings contain a subset of the inputs that constitute the recording; different types of tests use different subsets of a recording's inputs. A subset is necessary when deriving a test: if the full set of inputs were used, then the test would become fully deterministic, making behavior changes impossible. The choice of subset also determines what types of nondeterminism become possible. For example, a test generation tool might use a recording's user inputs (mouse events, scroll events, keyboard events, etc.) to synthesize a user interface test, but permit different program versions or event loop schedules. A tool that tests the throughput of a rendering engine could replay a recording as fast as possible, while permitting nondeterministic execution that does not cause the executing web program to diverge. A tool for minimizing a failure recording could discard inputs which are not causally related to a failure.

Below, I summarize some of the open research problems and promising approaches for generating different types of tests from deterministic replay recordings.

---

[1]STS is a random input generator and simulator that finds bugs in software-defined networks (SDN). It uses deterministic replay to find failure-revealing event interleavings, and to explain the causal sequence of events responsible for a failure.

8.2.1 User Interface Tests

Some of the most challenging tests to write are those that exercise interactive and dynamic behaviors. User interface (GUI) tests are difficult to author because the functionality and interfaces that they exercise are highly visual and change frequently. GUI tests also difficult to maintain: if tests are not written in a robust way, they will begin failing whenever anything substantial changes in the user interface. For example, a test that encodes user input events as a series of absolute screen coordinates is straightforward to create, but may fail if a different element is positioned at that coordinate during a later test run. To make tests less brittle to changes, prior work has investigated strategies for robustly identifying interface elements across program versions using visual [51, 178] or structural [14, 86, 176] properties. With additional research, these techniques could be used to synthesize robust GUI tests from a captured recording.

8.2.2 Performance Regression Testing

Detecting performance regressions is critical when making changes to a browser engine or JavaScript runtime. Browser vendors are especially interested in two performance metrics: latency and throughput. In the browser, *latency* (or time-to-first-paint) is the time required to download, parse, and render a web program to the screen. *Throughput* captures the rate over time at which a browser engine can respond to user actions and execute JavaScript code.

Measuring a browser's latency is fairly easy because loading web programs does not require user interaction. In contrast, measuring a browser's throughput is difficult because some of the most important use cases are also the most challenging to exercise with automated tests. Automated tests cannot easily recreate what occurs when a user uses an interactive application such as Facebook, Gmail, or Google Docs. Server responses are nondeterministic and may change over time; even if network traffic is simulated using a replaying reverse-proxy [138, 148], nondeterministic JavaScript and DOM APIs can easily

cause successive test executions to behave very differently. Due to the difficulty of testing these interactions, browser vendors today rely on "non-interactive" DOM and JavaScript benchmarks[2]. These short-running web programs are not representative of end-user browsing activity [138], but are better than micro-benchmarks or no benchmarks at all.

What if realistic performance tests could be synthesized from an execution captured by Dolos? In theory, this should be as straightforward as replaying a recording on different browser engine versions and collecting performance measurements. For example, browser vendors often want to track execution times for specific API calls, processing times for event loop tasks, overall memory usage, garbage collector activity, etc. In practice, re-executing a replay recording across different rendering engine versions is effectively an open research problem. Browsers gain and lose functionality frequently, and these differences may cause a replayed execution to diverge. Web programs often use dynamic feature detection—testing at runtime whether an API is implemented—when deciding which code path or library implementation to use. Thus, exposing a different set of platform features and APIs to a replayed web program may cause different code to execute. Even if publicly visible APIs do not change between browser versions, changes to browser architecture or policies can impact internal browser engine nondeterminism and cause divergence. For example[3], event loop tasks could be scheduled differently in response to resource contention, resource cache policies could change, some work could be moved to its own thread or process, and so on.

A different approach to creating performance tests is to synthesize a best-effort deterministic test that does not rely on a deterministic replay infrastructure. Richards et al. pioneered this approach with JSBench [138], a tool for generating JavaScript test scripts that approximate a captured browsing session. These tests are (necessarily) not com-

[2]BrowserBench: `https:///www.browserbench.org/`

[3]The changes mentioned here have occurred in upstream WebKit over the past few years during the course of my research. Each required nontrivial adjustments to the record/replay hooks used by Dolos.

pletely deterministic, but their performance characteristics are stable enough across multiple browser versions to be used as a performance metric. Access to more precise recordings may allow synthesized tests to be more consistent across browser engine versions.

8.2.3 Minimizing Recordings

Bug reports often contain irrelevant, misleading, or wrong information [184] that can hinder rather than help a developer in resolving an issue. The ability to submit a replayable recording produced by Dolos can reduce the amount of skill and effort required to report a bug (i.e., by writing reproduction steps, capturing output, or other steps). However, since recordings contain much more data than prose descriptions, recordings represent a potential *increase* in the amount of irrelevant information that a developer must investigate. Ideally, a recording submitted in lieu of reproduction steps would only contain the specific interactions and events necessary to reproduce a failure. But, even if a user creates a recording of themselves following reliable reproduction steps, they will unintentionally create user inputs—such as unhandled `mousemove` events—that become part of the recording but are not necessary to reproduce a failure.

Like program slicing and other techniques to aid comprehension, test case minimization techniques operate on the assumption that shorter recordings are strictly more useful because they reduce the amount of information that must be processed. This assumption seems reasonable, but it would be prudent[4] to obtain evidence to support this hypothesis before spending significant effort to develop minimization techniques whose usefulness is on it. In ongoing research outside the scope of this dissertation, my collaborators performed feasibility studies [67] to test whether this assumption is true in practice. We found that when traditional user interface test scripts [163] were minimized by removing irrelevant inputs, participants were able to diagnose and fix faults more quickly and successfully.

[4]Many reasonable assumptions in programming languages research have been debunked with just a few months' of curiosity and skepticism. For example, contrary to widely-held assumptions, recent studies

By thinking of a deterministic replay recording as an input file to a rendering engine, recordings can be shortened by applying well-known delta debugging algorithms [111, 182] that are often applied to the test case reduction problem [134]. Pruning irrelevant operations from a deterministic replay recording is inherently difficult because the strict invariants of fully deterministic execution are fragile. In the general case, dependencies between user inputs, network callbacks, and program data are extremely difficult to reason about using traditional program slicing techniques[5]. Scott et al. [147] have investigated a simpler alternative based on delta debugging and re-executing to detect divergence. First, a specific input and its dependencies are removed from a recording based on domain knowledge or observations of coarse-grained causal dependencies. For example, Dolos could observe the relationship between an executed event loop input and the later event loop inputs that it enqueues, and use these dependencies when removing inputs. This approach is unsound because it does not consider fine-grained data dependencies in JavaScript code, but any divergences caused by disrupted data dependencies can be easily detected by re-executing the reduced recording.

When reducing a recording, preserving a particular failure or behavior is often more important than maintaining strict determinism. For example, if a developer is only interested in the buggy behavior exhibited by a search bar widget, then inputs that affect other widgets on the page can be discarded. In this scenario, a recording reduction tool heavily depends on finding effective *test oracles* to guide delta debugging or other minimization algorithms. Finding effective, automated oracles for interactive behaviors and visual outputs is an active area of research [176].

---

have found that web programs frequently use dynamic and reflective language features [137], and a list of suspicious statements does not help much in isolating a fault [124].

[5] At the time of writing, I am not aware of any research prototypes that can usefully trace data and control dependencies through large-scale multi-process software systems such as browsers or operating systems. If such a prototype existed and could identify irrelevant inputs, it would still be necessary to validate the removal via re-execution.

8.3 On-demand, Retroactive Dynamic Analysis

The availability of deterministic replay could make many heretofore impractical program comprehension tools feasible for use with realistic programs and tasks. In his dissertation [42], Cornelissen identifies key intrinsic benefits and drawbacks of dynamic analysis for program comprehension:

1. Benefit: *Enhanced precision* with respect to actual software behavior, such as executed control flow paths or call-site dispatch targets.

2. Benefit: The possibility for *goal-oriented strategies* which only analyse specific execution behavior: namely, features exercised through an execution scenario provided by the user.

3. Drawback: The analysis is inherently *incomplete*, only covering the small fraction of possible executions and runtime states covered by an execution scenario.

4. Drawback: It is difficult and important to find appropriate *execution scenarios* over which the analysis should be run.

5. Drawback: *Scalability* is a major concern due to the large amounts of data collected, stored, analyzed, and presented to the user.

6. Drawback: *Observer effects* can cause an instrumented program to no longer reproduce a behavior of interest, especially when the program uses multithreading or depends on precise execution timings.

The research literature contains thousands of papers describing dynamic analysis techniques which are infeasible for regular use because the drawbacks—especially *scalability*—outweigh the benefits. Most dynamic analysis techniques assume that an execution

is transitory, so as the execution proceeds, an analysis must pessimistically collect all potentially relevant runtime state. However, excessive data collection is very frequently the limiting factor for scalability. For example, the Whyline for Java [82] can only capture execution scenarios up to a few minutes in length before the generated trace completely fills main memory. Scalability suffers because an analysis may add significant runtime slowdowns, consume too much main memory, disk space, network traffic, or induce observer effects. Applying principles from modern distributed systems and databases can increase the throughput for trace storage [131] and analysis [130], but this approach requires extensive engineering effort.

Deterministic replay could shift the benefit/drawback situation of dynamic analysis techniques to the point that many such techniques become feasible for realistic tasks and programs. This could be accomplished by several means: moving data collection and analysis in time in space; avoiding observer effects; improving completeness by combining executions; and designing entirely new multi-stage dynamic analysis techniques.

8.3.1 Improving Scalability and Reliability

The most straightforward improvement to scalability is the potential to decouple data collection and analysis from a specific execution [38, 39] or computing resource. This has clear usability benefits: a user can first capture an execution scenario using deterministic replay, and later perform the expensive instrumentation and analysis only if it is truly necessary. A user can run a dynamic analysis over a captured execution on a remote machine with more computing resources. (This is essentially the reverse of the solution proposed above for debugging field failures.) The initial recording phase incurs minimal overhead from deterministic record/replay, and subsequent re-executions—which are both deterministic and instrumented—are guaranteed to avoid certain observer effects.

The deterministic nature of Dolos and other record/replay infrastructures eliminates entire classes of observer effects. In particular, those caused by task interleavings or differ-

ent timings are impossible because these aspects of execution must be carefully controlled by the infrastructure in order to achieve deterministic execution. Most observer effects caused by deterministic replay are related to the set of features that are unsupported or disabled during capture or playback. For example, Dolos disables certain in-memory and OS-level resource caches during capture and replay. Many replay infrastructures that operate at the level of POSIX system call [113, 145] or a virtual machine [170] force all threads to execute on a single core during capture and replay. A dynamic analysis' instrumentation itself could be the source of some observer effects. For example, re-executing a recording with an instrumented browser may slow down an execution due to the extra memory usage and larger code size associated with instrumentation. If the instrumentation is transparent—that is, it doesn't diverge execution and is undetectable by client code—then observer effects caused by instrumentation are limited to runtime-related measurements.

8.3.2 Making Dynamic Analysis Interactive

A less understood direction for research is how deterministic replay enables new dynamic analysis techniques. Existing dynamic analysis techniques must pessimistically capture any potentially relevant data at runtime, on the assumption that executions are transitory. Some tools such as Whyline [82] initially collect runtime data in bulk, then perform post-hoc, on-demand data analysis to drive interactive user interfaces. What if data collection could be performed on-demand? How can a dynamic analysis collect data iteratively as it is needed, rather than all at once? How do user information needs map to configurations of instrumentation or an analysis?

To illustrate the potential of on-demand data collection, consider a scenario where a developer wants to find hot interprocedural execution *paths* and improve their performance. Existing tools provide complimentary capabilities that are useful in this scenario, but uses of these tools are mutually exclusive and require preemptive deployment. A tracing profiler [3] captures an exact calling context tree, but induces high runtime over-

head that makes it unsuitable for profiling hot code. A sampling profiler [61] has very low runtime overhead, but only reports hot call stacks, obscuring paths through functions and fast or infrequently-executed code. Branch and path profilers [9] are designed to capture frequently executed sequences of basic blocks, but are even slower than tracing profilers.

Using iterative data collection, it would be possible to build a developer tool that mixes and matches the capabilities of these profilers to quickly identify hot interprocedural paths within a captured execution. Such a tool can accelerate a developer's workflow by only turning on profilers as necessary, and by configuring each profiler invocation to discard irrelevant information. First, the tool re-executes with a sampling profiler enabled to detect methods that dominate execution time. To identify related functions in the call graph that run quickly or are infrequently executed, the tool re-executes again using a tracing profiler. Finally, to detect the paths that intersect with user-selected functions or branches, the tool re-executes once more with a path profiler configured to discard paths that do not intersect the program point of interest. In effect, this hypothetical tool can achieve the goals of adaptive [20] or "bursting" profilers [183]—only profile code that's interesting—by composing multiple uses of well-understood single-purpose profilers on successive playbacks.

8.4 A Database of Reusable Executions

Deterministic replay can enable large-scale corpus analysis of web program executions. Researchers and browser vendors frequently ask questions about how web programs behave at runtime: which browser and language features they use, whether optimizations pay off in practice, and whether a policy change would break existing web programs. In many cases, these questions can only be answered satisfactorily by analyzing many executions of real-world web programs. Recent research projects such as WebZeitgeist [87] have demonstrated the feasibility and utility of "mining" design elements from a corpus of thousands of static web pages. What if it were possible to mine *dynamic behaviors* from a corpus of thousands of captured executions?

Recent efforts to standardize the next version of JavaScript illustrate the importance of wide-scale impact analysis. In one instance[5], several standards committee members disagreed as to whether introducing a new method, `Array.prototype.contains`, would break web programs that make use of MooTools[5] (a popular JavaScript utility library). At the time, MooTools was used by 6% of all public web domains. Committee members were vexed: a new method in JavaScript's standard library should not potentially break millions of web programs, nor can they fix a library deployed on 3rd-party web servers, nor could they obtain real data as to whether this hypothetical corner case was an actual issue. In the end, the standards committee conservatively renamed the `contains` methods in `String` and `Array` to `includes`, breaking with the naming convention used by other new data types `Map` and `Set` and standard libraries in other languages.

With existing browser technology, the only way to answer many of these questions with runtime data is to manually interact with web programs using an instrumented browser. In prior work [137, 139] that analyzed whether and how often web programs used dynamic language features, my co-authors and I manually collected over 10,000 traces by browsing hundreds of web programs using a specially-instrumented browser. This took over 2 man-weeks of effort to produce results for each paper. This manual approach is also very brittle: all previously gathered traces are rendered useless if instrumentation was incorrect or missing. It also harms reproducibility of research results: in this model there is no way for someone to verify that the traces correspond to reasonable user interactions. The same corpus cannot be reused across multiple research projects, resulting in related but incomparable results.

If a corpus of executions is captured as recordings using Dolos, then trace data could be regenerated as needed by replaying recordings using an instrumented browser. Traces can be produced via any appropriate mechanism, such as through an extension, injected page content, wrapping a browser view inside a tracking harness, or by instrumenting

[5] `Array.prototype.includes` on GitHub: `https://github.com/tc39/Array.prototype.includes/`

[5] MooTools: `http://mootools.net/`

the browser engine itself. Traces produced by manual interaction or deterministic re-execution should not significantly differ, with some exceptions. For example, a trace may differ if: instrumentation causes execution to significantly diverge (Section 4.3); the execution depends on unsupported platform features (Section 3.4.4); or the trace captures aspects of execution that are intentionally left nondeterministic (Section 3.4.1).

Automatically generating executions that are representative of real user interactions and browsing behavior is an active area of research [108]. However, even without automatic generation, using Dolos recordings as a canonical representation of execution factors out repetitive interactions, making collection a one-time cost. With some improvements to how recordings are shared, versioned, and anonymized (Section 8.1), the task of capturing manual interactions could be distributed to any population of consenting browser users. Instead of performing analysis-specific data collection on an end-user's machine (as proposed by Liblit [102]), data is collected on-demand by re-executing an end-user's browsing session.

Chapter 9

CONCLUSION

Many developers have longed for the magical ability to go back in time when debugging a frustrating failure in a program. This dissertation investigates two lines of research that enable the vision of going back in time to understand program behavior.

The first half of realizing this vision is the capability to capture and revisit past program states. The first half of the dissertation describes how a fast, transparent, and pervasive deterministic replay infrastructure can be used to capture and extract program states. Chapter 3 presents Dolos, a novel deterministic replay infrastructure that targets the highly dynamic, visual, and interactive domain of web programs. Dolos is the first such infrastructure to adapt precise, low-overhead virtual machine replay techniques to modern browser runtimes. Chapter 4 further develops several additional Dolos features for extracting program states, detecting errors, and navigating to past program states.

The second half of realizing this vision requires the development of retroactive tools for finding and using program states. The second half of this dissertation describes several such retroactive tools that support different strategies for navigating through a captured execution. Section 5.2 describes Timelapse, a user interface for visualizing and navigating a captured execution according to its input events. Section 5.3 introduces the concept of *data probes* and *time-indexed events* as affordances for navigating through a captured execution via its logged runtime states. Chapter 6 describes Scry, a feature location tool that enables a developer to navigate to past program states via their visual outputs. tScry is a powerful approach for finding behavior-relevant program states without requiring significant code familiarity.

This dissertation has also been an adventure in how to move beyond the status quo of

uninspired, limited developer tools and unrealistic, untested research tools. On the user interface side, this dissertation provides examples of both successful and less-promising approaches to designing retroactive tools that solve real use problems. Scry in particular was the one tool that had great traction, was well-scoped in its goals, and could clearly benefit from deterministic replay (but not become useless without it). I am convinced that the way to proceed is to develop more such task-specific tools, and integrate their capabilities with deterministic replay when it makes sense to do so. On the technical side, deterministic replay has been a decent research success and a great industrial success. I am confident that deterministic replay will serve as the foundation for many more applications to software engineering outside of program understanding tools (Chapter 8).

This dissertation provides a glimpse into what is possible when deterministic replay capabilities are pervasive, transparent, and integrated into the runtime platform itself. In the course of performing the research described by this dissertation, I have demonstrated the potential of replay infrastructures for program comprehension so that their value is more widely appreciated by platform and tool developers.

Appendix A

RESEARCH PROTOTYPES AND DEMOS

As a tool- and prototype-centric dissertation, much of this work's intermediate output and external visibility is in the form of code, prototypes, and interactive demos. This section lists the major prototypes and demos developed as part of this dissertation. For each, I briefly explain its purpose, research results derived, how it relates to other repositories or prototypes, and where the code may be obtained.

A.1 Prototypes

A.1.1 Pre-Timelapse (2010–2011)

Early experiments in tracing and replaying web content took place in a branch of the DynJS [140] fork of WebKit in May 2011. Upstream changes were occasionally merged to the branch in large batches. I performed merges less and less frequently due to DynJS's invasive instrumentation of the JavaScript interpreter and increasing complexity of the JavaScript runtime. Once it became clear that deterministic replay was the platform on which other research results would be built, I created a new repository for the development of deterministic replay techniques.

A.1.2 `timelapse-hg`[1] *(2011–2012)*

[1] `timelapse-hg` on Bitbucket: `https://www.bitbucket.org/burg/timelapse/`

A new fork of WebKit, initially codenamed Timelapse[2], was created in Autumn 2011. This repository[1] contained a clean reimplementation of previous deterministic replay experiments, and was the basis for the first (rejected) conference submission describing Timelapse. This prototype was the first to save nondeterministic inputs to a file, integrate with the Web Inspector interface, and shares many of the same architectural choices as more recent prototypes. Its interface (Figures A.1 and A.2) was the basis for the Timelapse interface, both in implementation details and interface design. The interface featured an overview with a row of timelines per input category, and a sortable table containing details for each input.

Unlike in later designs, this early prototype dispatched event loop inputs preëmptively and asynchronously. Similar to instruction-counting techniques used in VM research [113, 145], its replay infrastructure counted the number of dispatched DOM events to know when to preëmptively inject the next event loop input. An asynchronous design was required to support the network replay machinery, which was an out-of-process reverse network proxy that captured and replayed browser network traffic. After months of dealing with unexplainable divergence and deadlock bugs, I abandoned both of these approaches in later prototypes in favor of the easier-to-debug synchronous event loop input dispatch scheme described in Section 3.2.2.

A.1.3 `timelapse-git`[3] *(2012–2013)*

The second major prototype was reimplemented from scratch as a fork of WebKit, this time in a git repository[3] based on the official WebKit git mirror[4]. This prototype was the basis for the paper published at UIST [30]. This prototype was used for the user studies in Chapter 7, so it received a lot of interface refinement and bug fixes. The visualization was

[2]To distinguish the separate technical and design contributions, I later gave the Timelapse moniker to the user interface, and named the replay infrastructure Dolos. Replay functionality in mainline WebKit is referred to as *Web Replay* in design documents [29] and `WEB_REPLAY` in code.

[3]`timelapse-git` on GitHub: `https://www.github.com/burg/timelapse/`

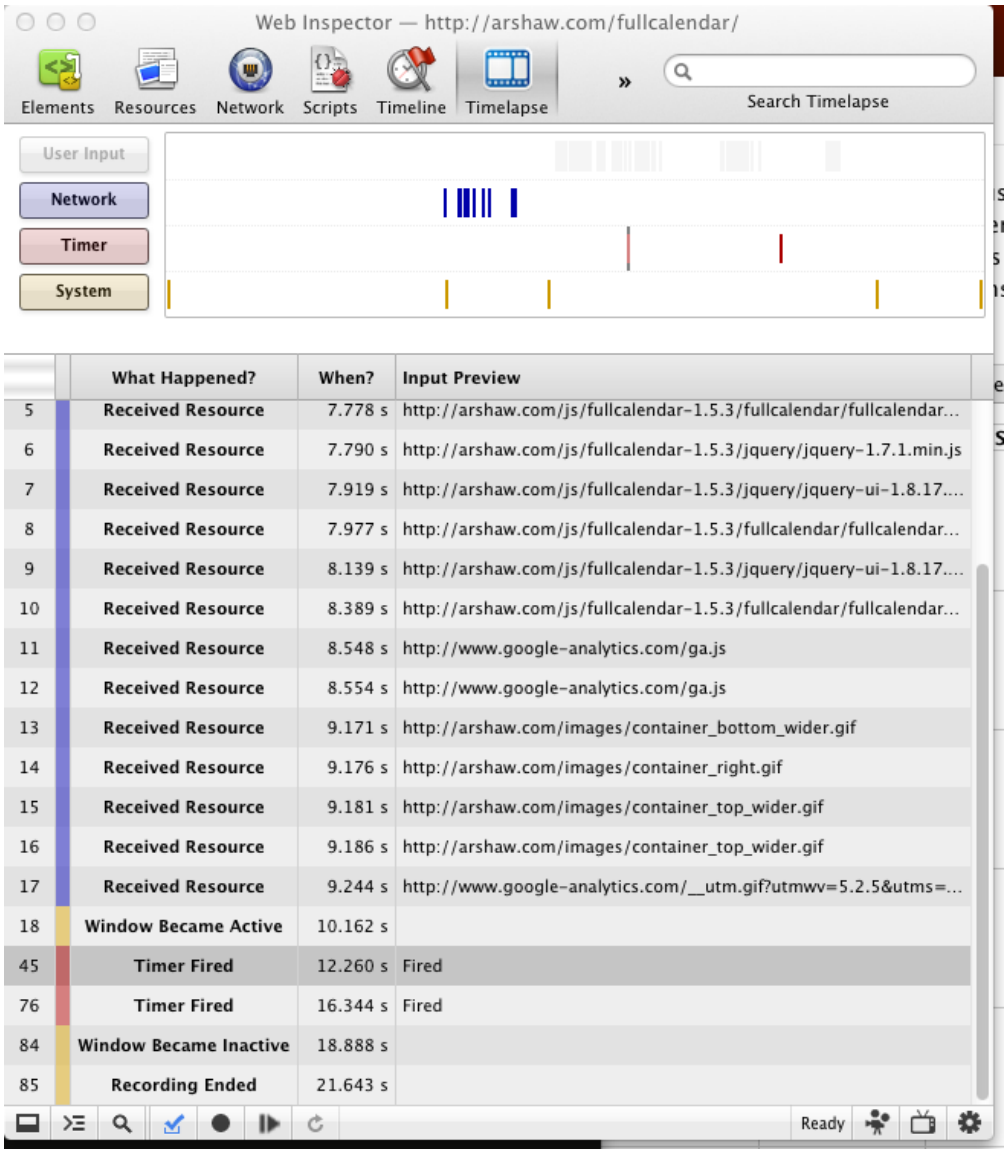

Figure A.1: An early prototype of Timelapse's input-centric timeline visualization. In this screenshot, the user has toggled the `User Input` button so that user input events are hidden from the table.

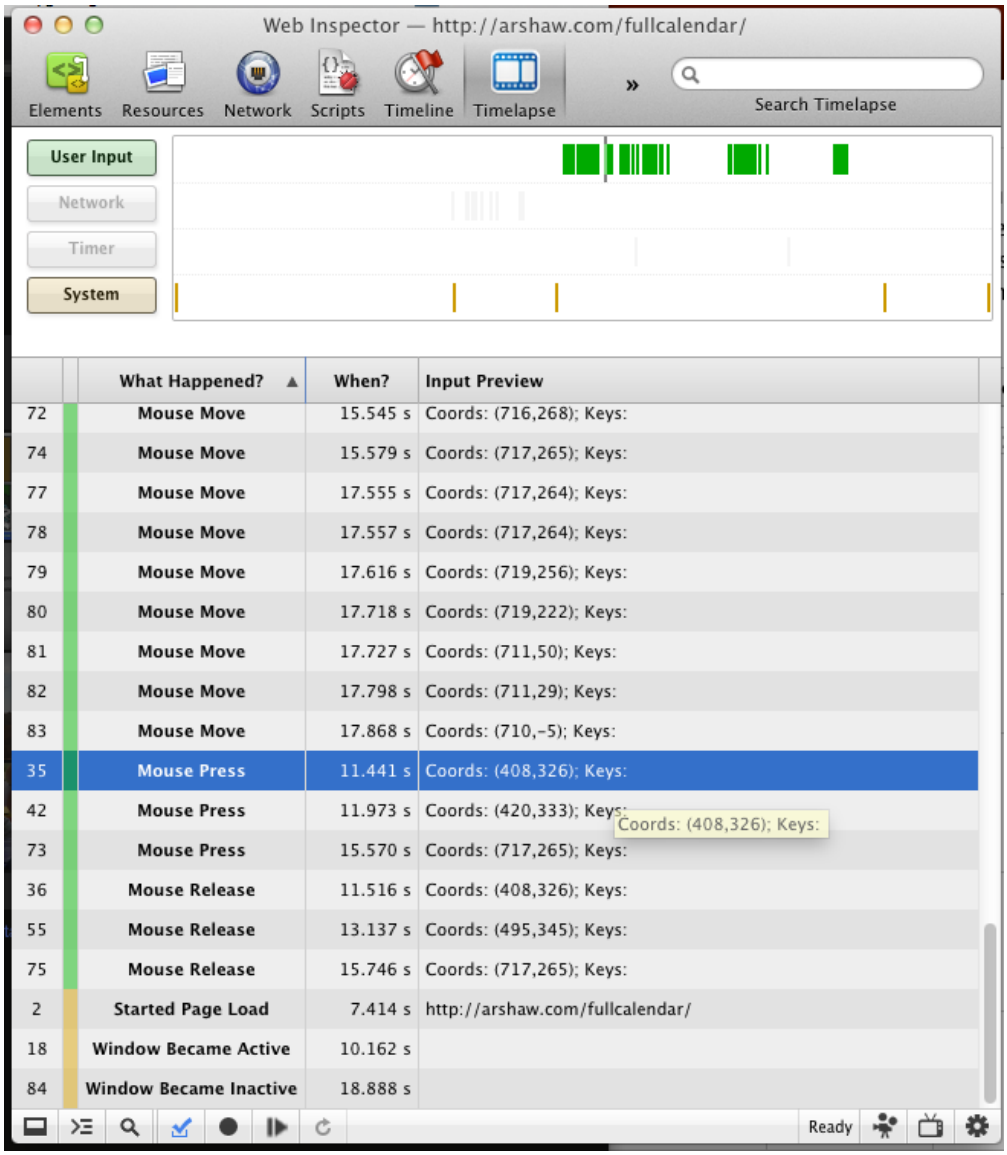

Figure A.2: An early prototype of Timelapse's input-centric timeline visualization. In this screenshot, the user has toggled the `Network` and `Timer` button and sorted the table of inputs by input type.

refined to more clearly show bursts of activity and scale to longer recordings. The timeline (Figure 5.1) encodes input density over time using bubble radius and fill opacity, with higher intensity shown as larger and darker bubbles. The interface gained an overview timeline that showed relative composition of input types and provided zooming capabilities. In addition to these refinements, many features and buttons were added to support scanning (Section 5.2.4), bookmarks (Section 5.2.3), and other integrations described in Section 4.2.

A.1.4 Interface Redesign (2013)

The third major prototype was based on the same replay infrastructure but featured a redesigned interface (Figure 5.5) to match the redesigned Web Inspector. This prototype was the basis for probes and time-indexed outputs described in Section 5.3. This new design abandoned the previous stacked input timeline visualization in favor of a minimalist line graph timeline overview and integration with the Web Inspector's existing timeline interface.

A.1.5 `replay-staging`[5] *(2013–2015)*

The last major prototype[5] focused on engineering and process improvements that made it easier to incorporate replay functionality into the upstream WebKit repository. Rather than maintaining a fork of WebKit, the `replay-staging` repository consisted of a patch series and a base WebKit commit. Instead of merging mainline changes into the fork, the patch series was rebased on top of new WebKit commits as functionality was contributed upstream. This prototype was the basis for the infrastructure extensions described in Chapter 4.

While I interned at Apple Inc. in early 2014, over 100 patches were developed in

[4] WebKit `git` mirror: `git://git.webkit.org/WebKit.git`

[5] `replay-staging` on GitHub: `https://www.github.com/burg/replay-staging/`

this repository and incorporated into mainline WebKit [28]. These changes included the core replay infrastructure from Chapter 3, a code generator for the input classes and file format used by Dolos, a testing harness for the Web Inspector, and general reliability and debugging improvements to the replay infrastructure (partially described in Chapter 4). I also implemented and shipped data probes (Section 5.3) as a new type of breakpoint action that does not depend or integrate with deterministic replay.

A.1.6 `scry-staging`[6] *(2014–2015)*

The Scry prototype was developed as a separate patch series[6] on top of WebKit, similar to `replay-staging`. Scry was initially built on top of the `replay-staging` patch series to demonstrate that its capabilities could be integrated with Dolos. When this was achieved, Scry was moved to its own repository to make it easier to rebase on top of newer WebKit commits. This prototype was the basis for the tool described in Chapter 6. None of its functionality has been incorporated into WebKit at the time of writing.

A.2 Demos

Each retroactive developer tool described in this dissertation required a significant amount of visual and interaction design work. Conveying the resulting "feel" of these tools is very difficult using prose. A live demonstration of the tool is much more compelling and can quickly clarify misconceptions caused by abstract text.

While this dissertation's contributions are all embodied in interactive prototypes, with current tools [64] it is infeasible to create archival-quality packages of these software on Mac OS X, the primary development platform for WebKit. In particular, WebKit has dependencies on public and private system libraries that are incompatible with other Mac OS X operating system versions. In lieu of software, I produced short videos to

[6] `scry-staging` on GitHub: `https://www.github.com/burg/scry-staging/`

demonstrate each major user interface: Timelapse[7], Probes and Time-indexed Outputs[8], and Scry[9].

[7]Timelapse Demo Video: https://www.youtube.com/watch?v=WYBPfNW2gsI

[8]Probes Demo Video: https://www.youtube.com/watch?v=ht3ckkNh6qM

[9]Scry Demo Video: https://www.youtube.com/watch?v=NENGl09Xq0I

BIBLIOGRAPHY

[1] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, 2014. (Cited on pages 16, 22, 24, and 105.)

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009. (Cited on page 11.)

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI 1997, Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997. (Cited on page 124.)

[4] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience report: Developing the Servo web browser engine using Rust. *CoRR*, abs/1505.07383, 2015. URL `http://arxiv.org/abs/1505.07383`. (Cited on page 30.)

[5] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN'11: The 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011. (Cited on pages 10 and 81.)

[6] 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. In *Proceedings of the 11th International Conference on Software Composition*, 2012. (Cited on page 17.)

[7] K. Arya, T. Denniston, A.-M. Visan, and G. Cooperman. Semi-automated debugging via binary search through a process lifetime. In *PLOS '13: Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, Farmington, PA, USA, 2013. (Cited on page 13.)

[8] T. Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999. (Cited on page 14.)

[9] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1996. (Cited on page 125.)

[10] E. T. Barr and M. Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2014)*, 2014. (Cited on pages 10, 43, and 44.)

[11] J. Bartlett. A nonstop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, 1981. (Cited on page 10.)

[12] BCEL Contributors. Apache Commons BCEL, 2014. `http://commons.apache.org/proper/commons-bcel/`. (Cited on pages 15 and 16.)

[13] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *USENIX 9th Symposium on OS Design and Implementation*, 2010. (Cited on pages 10, 31, and 32.)

[14] G. L. Bernstein and S. R. Klemmer. Towards responsive retargeting of existing websites. In *Proceedings of the Adjunct Publication of the 27th ACM Symposium on User Interface Software and Technology*, 2014. (Cited on page 118.)

[15] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *The 2010 ACM Conference on Computer Supported Cooperative Work*, 2010. (Cited on page 116.)

[16] 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. In *The Second International Conference on Virtual Execution Environments*, 2006. (Cited on pages 11, 16, and 18.)

[17] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proceedings of the 3rd Conference of Security and Trust*, 2014. (Cited on page 17.)

[18] P. Bille. A survey on tree edit distance and related problems. *Theoretical Comput. Sci.*, 337, June 2005. (Cited on page 96.)

[19] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: language-independent program slicing. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, 2014. (Cited on pages 25 and 105.)

[20] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005. (Cited on pages 17 and 125.)

[21] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI 2000, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000. (Cited on page 13.)

[22] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. *SIGOPS Operating Systems Review*, 17(5):90–99, 1983. (Cited on page 10.)

[23] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Trans. Comput. Syst.*, 7(1):1–24, 1989. (Cited on page 10.)

[24] 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. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010. (Cited on page 22.)

[25] 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. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009. (Cited on pages 19, 20, and 80.)

[26] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995. (Cited on page 10.)

[27] P. Bright. The web is getting its bytecode: WebAssembly, June 2015. `http://arstechnica.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/`. (Cited on page 16.)

[28] B. Burg. Announcement: web replay support, 2014. `https://lists.webkit.org/pipermail/webkit-dev/2014-January/026062.html`. (Cited on page 135.)

[29] B. Burg. The mechanics of web replay, 2014. `http://trac.webkit.org/wiki/WebReplayMechanics`. (Cited on pages 44 and 131.)

[30] B. Burg, R. J. Bailey, A. J. Ko, and M. D. Ernst. Interactive record and replay for web application debugging. In *Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, 2013. (Cited on pages 24, 26, 52, 62, 81, 82, 105, 107, and 131.)

[31] B. Burg, K. Madonna, A. J. Ko, and M. D. Ernst. Interactive navigation of captured executions via program output. Unpublished Draft, 2014. (Cited on pages 52 and 62.)

[32] B. Burg, A. J. Ko, and M. D. Ernst. Explaining visual changes in web interfaces. In Review, 2015. (Cited on page 80.)

[33] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004. (Cited on pages 17 and 23.)

[34] T. Cardenas, M. Bastea-Forte, A. Ricciardi, B. Hartmann, and S. R. Klemmer. Testing physical computing prototypes through time-shifted and simulated input traces. In *Proceedings of the 21st ACM Symposium on User Interface Software and Technology*, 2008. (Cited on pages 10, 11, and 24.)

[35] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: How and why developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007. (Cited on page 20.)

[36] P. K. Chilana, A. J. Ko, and J. O. Wobbrock. LemonAid: selection-based crowdsourced contextual help for web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012. (Cited on page 24.)

[37] J.-d. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, 1998. (Cited on page 10.)

[38] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008. (Cited on pages 18, 105, and 123.)

[39] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with Crosscut. In *The 2010 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010. (Cited on pages 18, 105, and 123.)

[40] J. Clouse and A. Orso. Camouflage: Automated anonymization of field data. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, 2011. (Cited on pages 11 and 116.)

[41] 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. (Cited on page 9.)

[42] B. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. PhD thesis, Technische Universiteit Delft, 2009. (Cited on page 122.)

[43] R. Curtis and L. D. Wittie. BUGNET: A debugging sytem for parallel programming environments. In *ICDCS '82, Proceedings 3rd International Conference on Distributed Computing Systems*, 1982. (Cited on page 10.)

[44] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *SIGOPS Operating Systems Review*, 44:97–102, January 2010. (Cited on page 22.)

[45] R. DeLine and K. Rowan. Code Canvas: Zooming towards better development environments. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, 2010. (Cited on page 22.)

[46] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: industrial experience with the Code Bubbles paradigm. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, 2012. (Cited on page 22.)

[47] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):275–295, 2007. (Cited on page 15.)

[48] F. Détienne. *Software Design - Cognitive Aspects*. Springer-Verlang, 2001. (Cited on page 19.)

[49] 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. (Cited on pages 10 and 23.)

[50] 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. (Cited on page 9.)

[51] M. Dixon and J. Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010. (Cited on page 118.)

[52] 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. *SIGOPS Operating Systems Review*, 36(SI):211–224, December 2002. (Cited on pages 10, 31, and 107.)

[53] C. Eckert and M. Stacey. Sources of inspiration: a language of design. *Design Studies*, 21(5):523–538, 2000. (Cited on page 80.)

[54] B. Eich. JavaScript at 17, 2012. https://www.youtube.com/watch?v=Rj49rmc01Hs. (Cited on page 1.)

[55] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. SeeSoft—a tool for visualizing line oriented software metrics. *IEEE Transactions on Software Engineering*, 18:957–968, November 1992. (Cited on page 22.)

[56] Facebook. React: a JavaScript library for building user interfaces, 2015. https://facebook.github.io/react/index.html. (Cited on page 103.)

[57] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology*, 22(2), March 2013. (Cited on page 63.)

[58] GDB Contributors. Inferiors and programs, 2014. http://sourceware.org/gdb/onlinedocs/gdb/Inferiors-and-Programs.html. (Cited on page 10.)

[59] 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. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009. (Cited on page 23.)

[60] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, 2005. (Cited on page 18.)

[61] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13:671–685, 1983. (Cited on page 125.)

[62] T. Grossman, J. Matejka, and G. Fitzmaurice. Chronicle: capture, exploration, and playback of document workflow histories. In *Proceedings of the 23rd ACM Symposium on User Interface Software and Technology*, 2010. (Cited on page 13.)

[63] P. J. Guo. Online Python Tutor: embeddable web-based program visualization for CS Education. In *SIGCSE '13: Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 2013. (Cited on page 22.)

[64] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011. (Cited on page 135.)

[65] 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. In *USENIX 8th Symposium on OS Design and Implementation*, 2008. (Cited on pages 10 and 37.)

[66] A. Hadiyat and contributors. Esprima: ECMAScript parsing infrastructure for multipurpose analysis, 2015. http://esprima.org. (Cited on page 16.)

[67] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *ESEC/FSE 2013: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2015. (Cited on pages 11 and 120.)

[68] M. Haverbeke and contributors. Acorn: a small, fast, JavaScript-based JavaScript parser, 2015. `https://github.com/marijnh/acorn`. (Cited on page 16.)

[69] J. Heer, J. D. Mackinlay, C. Stolte, and M. Agrawala. Graphical histories for visualization: supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1189–1196, 2008. (Cited on page 13.)

[70] M. Hertzum and A. M. Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing & Management*, 36:761–778, 2000. (Cited on page 19.)

[71] M. Hilgart. Step-through debugging of GLSL shaders, 2006. (Cited on page 82.)

[72] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: towards a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7:174–196, 2000. (Cited on page 19.)

[73] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE 2010, Proceedings of the ACM SIGSOFT 18th Symposium on the Foundations of Software Engineering*, 2010. (Cited on page 32.)

[74] D. F. Jerding and J. T. Stasko. The information mural: a technique for displaying and navigating large information spaces. In *InfoVis '95: Proceedings of the First Information Visualization Symposium*, 1995. (Cited on page 22.)

[75] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, 2012. (Cited on pages 11 and 116.)

[76] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th ACM Symposium on User Interface Software and Technology*, 2011. (Cited on page 69.)

[77] J. Kato, S. McDirmid, and X. Cao. DejaVu: integrated support for developing interactive camera-based programs. In *Proceedings of the 25th ACM Symposium on User Interface Software and Technology*, Cambridge, MA, USA, 2012. (Cited on pages 10, 11, 13, and 24.)

[78] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97, the 11th European Conference on Object-Oriented Programming*, 1997. (Cited on pages 14 and 15.)

[79] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005. (Cited on page 9.)

[80] A. J. Ko. *Asking and answering questions about the causes of software behavior*. PhD thesis, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 2008. (Cited on page 26.)

[81] A. J. Ko and B. A. Myers. Designing the WhyLine: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2004. (Cited on page 105.)

[82] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Method-*

ology, 20(2):1–36, September 2010. (Cited on pages 12, 14, 16, 20, 21, 22, 23, 24, 25, 73, 82, 90, 105, 123, and 124.)

[83] 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. *IEEE Transactions on Software Engineering*, 32(12):971–987, December 2006. (Cited on pages 20 and 73.)

[84] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, 2007. (Cited on pages 19 and 116.)

[85] R. Konuru. Deterministic replay of distributed Java applications. In *International Parallel and Distributed Processing Symposium (IPDPS'00)*, 2000. (Cited on page 10.)

[86] R. Kumar, J. O. Talton, S. Ahmad, and S. S. Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011. (Cited on pages 81, 96, 103, and 118.)

[87] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. Webzeitgeist: Design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. URL `http://vis.stanford.edu/papers/webzeitgeist`. (Cited on page 125.)

[88] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocesor operating systems. In *ACM SIGMETRICS 2010: International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2010. (Cited on page 11.)

[89] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *ICSE'10, Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010. (Cited on page 20.)

[90] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, Revo, NV, USA, 2010. (Cited on pages 20 and 26.)

[91] T. D. LaToza and B. A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, Portland, OR, USA, 2011. (Cited on page 21.)

[92] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, Pittsburgh, PA, USA, 2011. (Cited on page 20.)

[93] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006. (Cited on page 19.)

[94] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008. (Cited on page 20.)

[95] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, February 2013. (Cited on pages 20, 63, 82, and 89.)

[96] G. Lefebvre. *Composable and Reusable Whole-System Offline Dynamic Analysis*. PhD thesis, University of British Columbia Department of Computer Science, Vancouver, BC, Canada, 2012. (Cited on page 18.)

[97] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore:

unifying source code and execution experience. In *EuroSys 2009*, 2009. (Cited on page 12.)

[98] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2009)*, 2009. (Cited on page 57.)

[99] B. S. Lerner. *Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, 2011. (Cited on page 52.)

[100] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, 2010. (Cited on page 15.)

[101] B. Lewis. Debugging backwards in time. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, 2003. (Cited on pages 12, 13, 14, and 21.)

[102] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of Wisconsin-Madison, 2004. (Cited on page 127.)

[103] T. Lieber, J. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada, 2014. (Cited on page 21.)

[104] J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: runtime migration of browser sessions for JavaScript web applications. In *Proceedings of the 22nd International World Wide Web Conference*, Rio de Janeiro, Brazil, 2013. (Cited on pages 43 and 44.)

[105] L. Marek, A. Villazón, D. Zheng, Yudi anad Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th*

International Conference on Aspect-Oriented Software Development, 2012. (Cited on pages 15 and 16.)

[106] 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. In *Proceedings of the 12th International Conference on Generative Programming and Component Engineering*, 2013. (Cited on page 15.)

[107] S. McDirmid. Usable live programming. In *Proceedings of Onward! 2013*, 2013. (Cited on page 13.)

[108] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):1–30, 2012. (Cited on pages 16, 24, and 127.)

[109] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for JavaScript applications. In *7th USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, USA, 2010. (Cited on pages 10, 16, 37, 63, 81, and 107.)

[110] M. Mirzaaghaei and A. Mesbah. DOM-based test adequacy criteria for web applications. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, 2014. (Cited on page 24.)

[111] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, 2006. (Cited on page 121.)

[112] Mozilla contributors. Mozilla Firefox web browser, 2014. `http://www.mozilla.org/en-US/firefox/desktop/`. (Cited on pages 10 and 45.)

[113] Mozilla Contributors. rr: lightweight recording & deterministic debugging, 2014. `https://rr-project.org/`. (Cited on pages 9, 10, 13, 32, 124, and 131.)

[114] Mozilla Contributors. The Narcissus meta-circular JavaScript interpreter, 2015. https://github.com/mozilla/narcissus/. (Cited on page 16.)

[115] Mozilla Contributors. Rhino: a JavaScript interpreter in Java, 2015. https://github.com/mozilla/rhino/. (Cited on page 16.)

[116] B. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions is user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006. (Cited on page 105.)

[117] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI 2007, Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, 2007. (Cited on pages 12, 15, and 16.)

[118] 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. In *Proceedings of the 23rd ACM Symposium on User Interface Software and Technology*, 2010. (Cited on pages 10, 11, and 24.)

[119] Nodejitsu, Inc. node-http-proxy: A fully-featured http proxy for node.js, 2015. https://github.com/nodejitsu/node-http-proxy/. (Cited on page 16.)

[120] R. O'Callahan. Efficient collection and storage of indexed program traces, 2006. http://www.ocallahan.org/Amber.pdf. (Cited on pages 11, 18, 23, and 82.)

[121] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, 2014. (Cited on page 16.)

[122] J. Odvarko and Firebug contributors. Firebug: web development evolved, 2013. http://www.getfirebug.com/. (Cited on page 10.)

[123] S. Oney and B. Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2009. (Cited on pages 10, 24, 37, 81, and 92.)

[124] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011. (Cited on pages 113 and 121.)

[125] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.53. URL `http://ipython.org`. (Cited on page 116.)

[126] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through interactivity: online analysis of run-time behavior. In *2010 17th Working Conference on Reverse Engineering*, 2010. (Cited on page 17.)

[127] 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, January 1998. (Cited on page 19.)

[128] D. Piorkowski, S. D. Fleming, I. Kwan, M. Burnett, C. Scaffidi, R. Bellamy, and J. Jordhal. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. (Cited on page 20.)

[129] G. Pothier and E. Tanter. Extending omniscient debugging to support Aspect-oriented programming. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, 2008. (Cited on pages 15 and 18.)

[130] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, Lancaster, UK, 2011. (Cited on pages 12, 18, 23, and 123.)

[131] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, Montreal, Canada, 2007. (Cited on pages 12, 14, 15, 18, 22, 23, and 123.)

[132] Project Contributors. Debugging your program using Valgrind gdbserver and GDB, 2014. `http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver`. (Cited on page 17.)

[133] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *WebApps'10: USENIX Conference on Web Application Development*, 2010. (Cited on pages 12 and 18.)

[134] J. Regher, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI 2012, Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, 2012. (Cited on page 121.)

[135] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys 2009*, 2009. (Cited on page 30.)

[136] S. P. Reiss and M. Renieris. JOVE: Java as it happens. In *ACM Symposium on Software Visualization*, 2005. (Cited on page 17.)

[137] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI 2010, Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010. (Cited on pages 12, 16, 17, 18, 121, and 126.)

[138] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2011)*, 2011. (Cited on pages 16, 24, 37, 118, and 119.)

[139] 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. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, 2011. (Cited on pages 16, 17, and 126.)

[140] G. Richards, J. Vitek, F. Z. Nardelli, A. Domurad, F. Meawad, C. Hammer, B. Burg, and S. Lebresne. Dynamics of JavaScript, 2015. `http://plg.uwaterloo.ca/~dynjs/`. (Cited on page 130.)

[141] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12): 889–903, December 2004. (Cited on pages 73 and 113.)

[142] D. Röthisberger. Querying runtime information in the IDE. In *Proceeding of the 2008 Workshop on Query Technologies and Applications for Program Comprehension*, 2008. (Cited on page 22.)

[143] D. Röthliberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *25th IEEE International Conference on Software Maintenance*, 2009. (Cited on page 15.)

[144] D. Röthlisberger. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, Universität Bern, 2010. (Cited on pages 21 and 86.)

[145] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'2005, Sixth International Symposium on Automated and Algorithmic Debugging*, 2005. (Cited on pages 37, 107, 124, and 131.)

[146] R. Salkeld, B. Cully, G. Lefebvre, W. Xu, A. Warfield, and G. Kiczales. Retroactive aspects: Programming in the past. In *WODA 2011: Ninth International Workshop on Dynamic Analysis*, 2011. (Cited on page 18.)

[147] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting

blackbox SDN control software with minimal causal sequences. In *SIGCOMM 2014*, Chicago, IL, USA, 2014. (Cited on pages 117 and 121.)

[148] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE 2013: The 9th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, St. Petersburg, Russia, 2013. (Cited on pages 15, 16, 18, 24, 104, 105, and 118.)

[149] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34:434–451, 2008. (Cited on pages 20, 26, and 69.)

[150] S. M. Srinivasan, S. Kandula, and C. R. Andrews. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004. (Cited on page 10.)

[151] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: a capture/replay tool for observation-based testing. In *ISSTA 2000, Proceedings of the 2000 International Symposium on Software Testing and Analysis*, 2000. (Cited on page 10.)

[152] M.-A. Storey. Theories, methods, and tools in program comprehension: Past, present, and future. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension*, 2005. (Cited on pages 19 and 24.)

[153] M.-A. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. In *Proceedings of the 5th Workshop on Program Comprehension*, 1997. (Cited on page 19.)

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

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2002. (Cited on page 22.)

[155] J. Stylos and B. A. Myers. Mica: a web-search tool for finding API components and examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006. (Cited on page 80.)

[156] M. Taeumel, B. Steinert, and R. Hirschfeld. The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. In *Proceedings of Onward! 2012*, 2012. (Cited on page 22.)

[157] K.-C. Tai, R. H. Carver, and E. E. Obaid. Deterministic execution debugging of concurrent Ada programs. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, 1989. (Cited on page 10.)

[158] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, 2003. (Cited on page 17.)

[159] E. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*, 2010. (Cited on page 17.)

[160] E. Tanter, I. Figueroa, and N. Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations, and applications. *Science of Computer Programming*, 80:311–342, 2014. (Cited on page 17.)

[161] Telerik. Fiddler web debugging proxy, 2013. `http://www.fiddler2.com/fiddler2/`. (Cited on page 24.)

[162] M. Terrano and P. Bettner. 1500 Archers on a 28.8: Network programming in Age of Empires and beyond. In *Gamasutra*, March 2001. `http://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php`. (Cited on pages 10 and 23.)

[163] The Selenium Project. Selenium WebDriver documentation, 2012. http://seleniumhq.org/docs/03_webdriver.html. (Cited on pages 10, 24, and 120.)

[164] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010. (Cited on pages 15 and 16.)

[165] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990. (Cited on page 13.)

[166] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *SoftVis '10: Proceendinsg of the 2010 ACM Symposium on Software Visualization*, 2010. (Cited on pages 22 and 23.)

[167] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *Communications of the ACM*, 40:38–43, April 1997. (Cited on page 24.)

[168] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013. (Cited on page 11.)

[169] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. URDB: a universal reversible debugger based on decomposing debugging histories. In *PLOS '11: Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, Cascais, Portugal, 2011. (Cited on page 13.)

[170] I. VMWare. Replay debugging on Linux, October 2009. http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf. (Cited on pages 10, 13, 32, 63, and 124.)

[171] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. In *CASE'93: Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering*, 1993. (Cited on pages 19 and 20.)

[172] W3C. WebAssembly community group, 2015. `https://www.w3.org/community/webassembly/`. (Cited on page 16.)

[173] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2002. (Cited on page 19.)

[174] WebKit contributors. The WebKit open source project, 2012. `http://www.webkit.org/`. (Cited on pages 35 and 101.)

[175] WebKit contributors. About Safari Web Inspector, 2014. `https://developer.apple.com/library/safari/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html`. (Cited on pages 10, 58, 65, and 101.)

[176] Q. Xie and A. M. Memon. Desiging and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16, February 2007. (Cited on pages 118 and 121.)

[177] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2), 2005. (Cited on page 25.)

[178] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22nd ACM Symposium on User Interface Software and Technology*, 2009. (Cited on page 118.)

[179] K. Yit Phang, J. S. Foster, and M. Hicks. Expositor: scriptable time-travel debugging with first-class traces. In *ICSE'13, Proceedings of the 35th International Conference on Software Engineering*, San Francisco, CA, USA, 2013. (Cited on page 18.)

[180] S. Yoo, D. Binkley, and R. Eastman. Seeing is slicing: Observation based slicing of picture description languages. In *Proceedings of the 14th International Working Conference on Soource Code Analysis and Manipulation*, 2014. (Cited on pages 25 and 105.)

[181] A. Zaidman, N. Matthijssen, M.-A. Storey, and A. van Deursen. Understanding AJAX applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218, April 2013. (Cited on page 44.)

[182] A. Zeller and R. Hildbrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 38(2), February 2002. (Cited on page 121.)

[183] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive context profiling. In *PLDI 2006, Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, Ottawa, Canada, 2006. (Cited on page 125.)

[184] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schrter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, September 2010. (Cited on pages 23, 26, and 120.)