Interactive Navigation of Captured Executions via Program Output

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

Computer Science and Engineering[1]
University of Washington
{burg, madonk, mernst}@cs.washington.edu

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

ABSTRACT

During debugging, maintenance and other comprehension activities, developers often try to reproduce and examine task-relevant program states. This common task requires significant manual effort using existing tools. Deterministic replay systems can capture and replay entire executions, but can only suspend execution at context switches and prior to user inputs.

We propose navigating an execution via its visual and text outputs as a means for quickly revisiting particular program states. We present *time-indexed outputs* as a mechanism for automatically reverting execution to specific outputs, and *data probes* as a user affordance for dynamically creating new time-indexed outputs. We present an algorithm for navigating to time-indexed outputs and discuss considerations for integrating these mechanisms into a production web developer tool.

Author Keywords

Programming, Navigation

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces

General Terms

Human Factors; Design

INTRODUCTION

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

Prior work has investigated deterministic replay techniques—which operate by controlling sources of nondeterminism at runtime—as a basis for automatically reproducing specific executions. Dolos and similar replay infrastructures [8] only provide affordances for navigating a captured execution by its user inputs, event loop tasks, or low-level signals. To suspend execution at a specific program point, a developer must isolate and replay to the preceding input and then use a breakpoint debugger to drive execution to a specific statement.

In prior work, researchers have observed [1, 9, 6] that developers greatly prefer to navigate executions by *outputs*, rather than by inputs. A program's outputs can correspond to relevant program states, and developers often work backwards from outputs when attempting to understand runtime behavior [5]. To this end, we contribute two extensions to previous replay systems that realize output-oriented navigation: *time-indexed outputs* and *data probes*.

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

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

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

EXAMPLE

Time-indexed outputs and data probes are designed to automate the tedious, error-prone tasks of suspending execution

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

and logging specific program states within a captured recording. This section uses a program maintenance task to demonstrate advantageous uses of probes and time-indexed outputs.

Color Picker[2] is a jQuery plugin that implements a color picker widget for RGB and HSV colorspaces. Karla, a developer, uses the widget in her web application. She is investigating a bug that manifests when a user manipulates the color picker's color component sliders (Figure 2). 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.

Karla starts by using a record/replay tool (such as Dolos [1]) to capture a recording that demonstrates the steps to reproduce the Color Picker bug. This recording makes further reproduction simple by allowing Karla to quickly jump to the input just prior to the failure, but she still must explore the thousands of lines of code that execute after this input using traditional debugging tools. She still must set breakpoints to suspend execution at specific lines of code, which is tedious because the program must be re-executed to test whether the breakpoints were positioned correctly. Logging program states is similarly laborious: to log color component changes, Karla has to edit the widget's source code, rebuild and re-deploy the website, capture a new recording, and then view the logged outputs.

Data probes and time-indexed outputs simplify Karla's investigation of the buggy interaction. To create a foothold for observing runtime behavior, Karla uses data probes to log RGB values as they change, since past component values are not stored or logged to the console. After using the built-in timelines view (Figure 1.a) to see what code executed during the recording, she guesses that the `moveIncrement` handler may contain the RGB values she wants to log. To see the effects of each drag event, Karla adds data probes before and after the handler modifies color components (at `colorpicker.js:127` and `colorpicker.js:131`, as seen in Figure 1.b). These data probes capture the value of the expressions `color.r`, `color.g`, and `color.b` (Figure 1.c) each time the associated line executes. Karla then replays the recording again to generate new probe samples. As the recording is replayed, the probe sidebar (Figure 1.c) begins to populate with probe samples. Looking at temporally-ordered probe samples in the console (Figure 3), 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 1.c and Figure 3). From that instant of execution, she can use the debugger to step into the event handler and work towards the root cause with the aid of actual runtime values. With time-indexed outputs and data probes, she's likely able to do this much faster and more systematically than with breakpoints and logging alone.

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

IMPLEMENTATION

In prior work, Burg et al. developed the Dolos deterministic record/replay infrastructure [1] for reproducing web application executions. Our prototype of data probes and time-indexed output is built on top of the Dolos infrastructure. Dolos, data probes, and time-indexed outputs have been incorporated into the open-source WebKit browser toolkit[3].

Creating and Evaluating Data Probes

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

The probes user interface supports comparing, relating, and navigating to probe samples. The probes sidebar (Figure 1.c) persists collected samples for all subsequent playbacks of the recording. This allows a user to stitch together a map of probe samples across the whole recording without necessarily replaying it contiguously with a specific set of data probes. The probes sidebar groups probe samples by call site to support comparisons. Probe samples are also printed to the 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.

Replaying to Output-Producing Statements

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

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

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

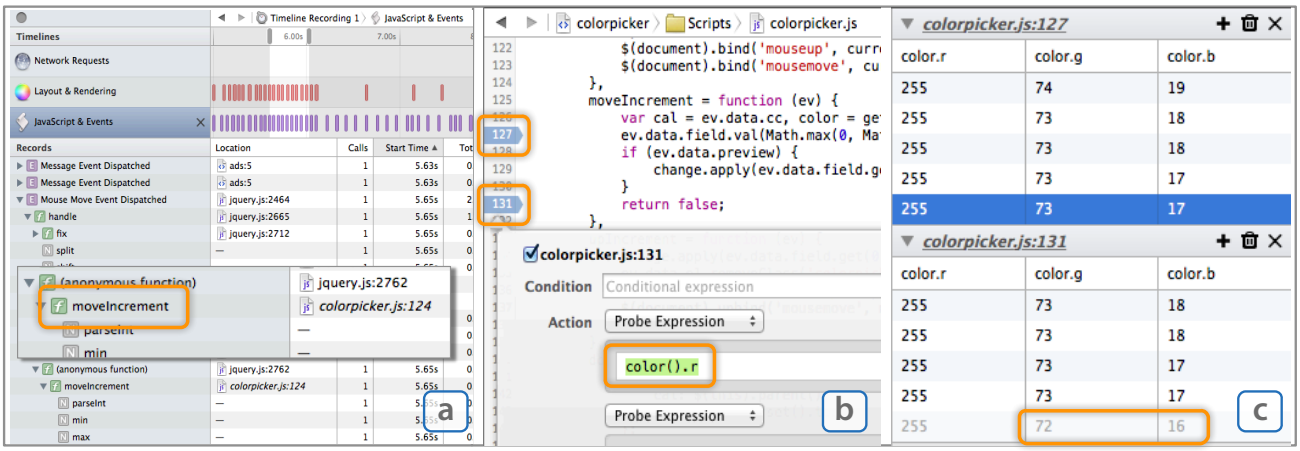

Figure 1. Using data probes to revert execution to relevant program states. In (a), Karla inspects the captured recording to find the `moveIncrement` drag event handler (highlighted). In (b), she adds two *data probes* (at lines 127 and 131) to log how color component values changed. In (c), she reverts execution directly to a probe sample (top, selected row) that immediately precedes erroneous component values (below, highlighted).

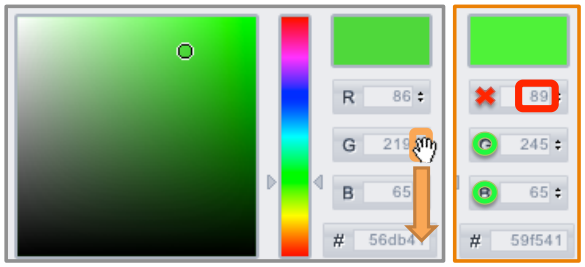

Figure 2. The Color Picker widget. When the G color component is adjusted downward (left), the R component unexpectedly changes (right).

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

- We can revisit time-indexed outputs without restarting playback from the beginning of a recording.
- We accumulate partial knowledge of time-indexed outputs across multiple discontiguous playbacks.
- We can automatically discover new probe samples (i.e., time-indexed outputs) in unknown sections of a recording.

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

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

Minimizing Breakpoint Use

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

RELATED WORK

Several lines of research investigate ways of tightening the feedback loop of editing code, looking at output, and debugging further. This section discusses those which use program output as beacons for navigating through an execution, and those which simplify the process of gathering runtime state.

Capturing and navigating executions

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

Two approaches—deterministic record/replay and post-mortem trace analysis—dominate research into capturing and navigating through executions. Deterministic replay research traditionally focuses on capturing executions with low overhead and high fidelity. Recent prototypes such as Aftersight [2] and Jalangi [10] perform dynamic analysis on-demand during replayed executions. Data probes also gather data from a replayed execution, but their placement is driven by user interaction rather than pre-defined analyses.

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

The Whyline for Java [5] is a heavyweight trace analysis tool that allows developers to ask why- and why-not questions about program output. It can reconstruct a program's visual output, textual output, and intermediate program state from the operation trace, and it uses program slicing techniques over the trace to answer program understanding questions on demand. By leveraging deterministic replay instead of post-mortem trace analyses, time-indexed outputs provides many of the same affordances as Whyline, but with near-zero runtime overhead and at interactive speeds on real web applications.

Live programming systems

The live programming paradigm [3] emphasizes tight feedback loops, typically by blurring or removing delays between editing a program and seeing effects of the changes. Live programming tools support quick iteration on inputs or the program itself, and often eschew imperative programming models to better support these goals. Below, we discuss two analogues to data probes and time-indexed outputs designed for other programming environments. While the affordances offered are similar, only data probes and time-indexed outputs are designed for the mainstream domain of imperative, interactive programs such as web applications.

DejaVu [4] supports interactive debugging and development of image processing algorithms. Similar to data probes, a user can add new debugging outputs (e.g. image filters, inferred skeletal models) which are automatically computed and juxtaposed with prior output on a timeline. Using the output timeline, the user can revert execution to a specific input or rendered output frame and inspect the program's state. Time-indexed outputs support a similar interaction for textual outputs, such as console logging or probe samples.

YingYang [7] is a programming environment that integrates live execution feedback with an emphasis on *traces* (similar to console output) and *probes* (similar to our probes). It supports rewinding execution to a logged output, and can substitute concrete values into the code/stack frame to explain the output's derivation. YingYang is built atop the Glitch live programming runtime. Glitch incrementally repairs program state as inputs or the program itself changes, so it has no notion of state over time, nor does it support temporally-ordered inputs. In contrast, our data probes and time-indexed output can be used to navigate recordings of imperative, interactive programs.

FUTURE WORK

Data probes and time-indexed outputs are a step towards the Whyline [5] vision of interactively investigating runtime behaviors. Data probes and time-indexed output provide a way to navigate captured recordings via console outputs and intermediate script states. In future work, we will continue to explore the Whyline vision while maintaining interactive performance and integration with production web development tools. We want to improve links between visual output and the code fragments that produced the output. We also plan to expand the scope of probes to include other program states, such as changes to the DOM tree structure or changes to an element's appearance or position.

ACKNOWLEDGEMENTS

The authors would like to thank the WebKit development team at Apple, Inc. for valuable discussions and code review.

REFERENCES

1. Burg, B., Bailey, R. J., Ko, A. J., and Ernst, M. D. Interactive record and replay for web application debugging. In *UIST* (2013).
2. Chow, J., Garfinkel, T., and Chen, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX* (2008).
3. Hancock, C. M. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.
4. Kato, J., McDirmid, S., and Cao, X. DejaVu: integrated support for developing interactive camera-based programs. In *UIST* (2012).
5. Ko, A. J., and Myers, B. A. Extracting and answering why and why not questions about Java program output. *ACM TOSEM 20*, 2 (Sep. 2010), 1–36.
6. Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE 32*, 12 (Dec. 2006), 971–987.
7. McDirmid, S. Usable live programming. In *Proceedings of Onward! 2013* (2013).
8. Mickens, J., Elson, J., and Howell, J. Mugshot: deterministic capture and replay for JavaScript applications. In *NSDI* (2010).
9. Robillard, M. P., Coelho, W., and Murphy, G. C. How effective developers investigate source code: an exploratory study. *IEEE TSE 30*, 12 (Dec. 2004), 889–903.
10. Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE* (2013).