# Addressing Misconceptions About Code with Always-On Programming Visualizations

**Tom Lieber**
MIT CSAIL
Cambridge, MA USA
tom@alltom.com

**Joel Brandt**
Adobe Research
San Francisco, CA USA
joel.brandt@adobe.com

**Robert C. Miller**
MIT CSAIL
Cambridge, MA USA
rcm@mit.edu

## ABSTRACT

We present Theseus, an IDE extension that visualizes run-time behavior within a JavaScript code editor. By displaying real-time information about how code *actually* behaves during execution, Theseus proactively addresses misconceptions by drawing attention to similarities and differences between the programmer's idea of what code does and what it actually does. To understand how programmers would respond to this kind of an always-on visualization, we ran a lab study with graduate students, and interviewed 9 professional programmers who were asked to use Theseus in their day-to-day work. We found that users quickly adopted strategies that are unique to always-on, real-time visualizations, and used the additional information to guide their navigation through their code.

### Author Keywords

Programming; debugging; code understanding

### ACM Classification Keywords

D.2.5. Software Engineering: Debugging Aids

## INTRODUCTION

Programmers are often wrong about what code actually does [5]. This causes them to generate incorrect hypotheses while reading, writing, and debugging code, to waste time and energy investigating false leads, and to introduce new bugs while modifying code [9, 16]. We hypothesize that programmers often have these misconceptions because code behavior is invisible most of the time. Few debugging interfaces are designed to be left on and visible during all phases of programming. Instead, the programmer must request information explicitly, for example by opening an inspector, setting a breakpoint, or inserting a print statement. This means that a faulty mental model is corrected only by applying it during debugging. We believe that programmers would benefit substantially from tools that *proactively* work toward correcting misconceptions.

Our solution is a code editor extension called Theseus (Figure 1). Theseus visualizes the program's run-time state using


**Figure 1. Theseus shows call counts for every function, and an asynchronous call tree allows the user to see how functions interact. In the log above, users can see which call to `fetch` corresponds to the failure without adding any debugging-specific code or instrumentation.**

code coloring and marginal notes, allowing the programmer to perceive that information unobtrusively as they read the code. A function body that has not been executed at all is shown with a gray background, and functions that have been called repeatedly are labeled with the call count. The colors and call counts update in real time so that the user can watch them respond to the actions they take in their program. Clicking on one of those call counts adds an entry to a log showing the arguments and return values of every call, retrieved from a program trace. Theseus organizes the log entries into a call tree that accounts for asynchronous invocations (such as event handlers), allowing programmers to quickly answer many time-consuming reachability questions [9].

In order to test whether these tools would help correct misconceptions more quickly, we ran two studies and deployed Theseus as an extension to the Brackets IDE[1]. Participants in the first lab study performed programming tasks with and

Submitted to CHI 2014

[1] `http://brackets.io`

without Theseus active. Theseus users identified the locations where chains of callbacks broke down more quickly (often within seconds of opening a file) than users with just a breakpoint debugger, and exhibited behaviors such as arranging their desktop so that they could see their code while interacting with their application.

In the second study, professional software developers used Theseus in their day-to-day programming activities. We then interviewed them to see how Theseus fit into their programming workflow.

The contributions of this paper are:

- An always-on, zero-click interface for displaying program execution information in the context of the programmer's source code. This interface helps the programmer keep their idea of how their code executes consistent with what is actually implemented.
- An event-oriented summary of program execution that enables one-click filtering of execution data and navigation of source code.
- An efficient approach for collecting, storing, and querying execution trace data that does not require modification of the runtime, and that works with multiple concurrent execution environments (e.g. both a client and server JavaScript virtual machine).
- Results from two studies which suggest that Theseus helps keep the programmer's idea of what their code does from diverging from what it actually does during development.

In this paper we will introduce related work, describe the interface and implementation, then discuss the results of our two studies.

## RELATED WORK

Theseus's interface was informed by research into the types of questions programmers ask while programming [9, 16], information foraging theory [3], as well as our observations of the tasks JavaScript programmers seem to find difficult or tedious. Some of the most relevant research, and its relationship to Theseus, is described in this section.

In his essay "Learnable Programming", Bret Victor outlines several of the features he believes a learnable programming environment and language should have, with control flow visualizations among them [17]. He presents an always-on interface that displays domain-specific representations of intermediate values alongside the code. DejaVu is a similar project whose domain is real-time video processing [7]. DejaVu associates the values of variables with the frame of video being processed so that users can inspect them by scrubbing to problematic points of the video. Widgets displaying those values are placed on a canvas among drawings and intermediate versions of the images being processed. Block-based programming environments like Scratch and Scheme Bricks highlight code as it executes, which can help users draw correspondences between the code and program behavior, though not after the code has finished [14, 6].

traceGL allows the user to see a zoomable visualization of the entire program trace, updated in real time [1]. After selecting a location in the trace, the user can can navigate the code, which is annotated colors to indicate what was executed, and examine the values of expressions by hovering over them with their mouse. Theseus's always-on visualizations are similar, but focused on summarizing all past behavior instead of inspecting individual slices of time.

Reacher [10] answers reachability questions by presenting a compact graph representation of the interactions between several functions. Reacher uses static analysis to generate the graph, whereas Theseus displays examples which have actually occurred. Thus, Reacher can answer questions about what *can* happen, and Theseus can answer questions about what the user has just *seen* happen. Theseus and Reacher provide complementary views of the same kind of information.

Whyline [8] is a debugging interface that users can ask questions like, "Why is this widget blue?" Whyline answers this question by generating a program slice of all the events that determined the line's color. The goal of Theseus is to make the programmer more aware of how their program is executing, even when there is no program output about which to ask questions.

The idea of adding edges to the call graph to aid debugging goes back to ZStep [11], an omniscient step debugger. A ZStep user could step to the point when a given expression was evaluated, to the next time the GUI changed, or to when a particular screen element was drawn. IntelliTrace [13] is similar in that it allows users to index into a program trace by selecting an event, such as a button click or an exception. In addition, some effort has been made to generate JavaScript stack traces that cross event boundaries [15], and more generally across client-server boundaries [12].

Timelapse offers self-contained, fully-replayable traces of entire web pages within WebKit browsers [2]. Theseus records program traces using source rewriting, which does not require modification of the virtual machine, but cannot perfectly recreate past machine states.

## THESEUS'S INTERFACE DESIGN

The design goal for Theseus is to put the results of program execution at the fingertips of the programmer. The programmer is intended to leave Theseus running during all stages of development—from empty file to finished product.

In this section, we demonstrate a few of the ways that Theseus's always-on visualizations allow the user to quickly perform sanity checks of their code's behavior, and how those same visualizations serve as warnings when the code starts to act differently than intended. Note how Theseus's visualizations allow the user to do all this without leaving the editor, adding any instrumentation, or even manipulating a debugging interface.

### Scenario: A Network Activity Indicator

In this scenario, Samantha is creating an animated widget to appear during network operations on an HTML page.

*A sanity check*

Samantha starts by sketching the functions she'll need: a function that creates an instance of the widget, called `Activity`, and a function to test the widget's operation as soon as the page loads.

After writing only the definitions of the widget constructor, and registering an empty callback function for when the page finishes loading, Samantha opens the page in a browser. The code coloring in her text editor immediately shows her that the code was parsed successfully (no syntax errors) and that the page load callback was registered and called correctly.

```
0 calls
    function Activity() {
20  }
21
1 call
    $(document).ready(function () {
23  });
```

When editing static HTML and JavaScript files, her IDE will automatically reload the browser after every save, so sanity checks like these are quick to make.

After writing some code to add the activity indicator to the DOM (Document Object Model), she saves the file and again gets immediate feedback that both functions were called and neither threw an exception.

```
1 call
    function Activity() {
20  }
21
1 call
    $(document).ready(function () {
23      $(document.body).append(Activity())
24  });
```

*Understanding timing*

Some time later, Samantha wants to update the DOM periodically to create the animated effect. Even before adding any code to the function to update the page, she can judge whether the timer is appropriately configured by how quickly the call counts change in the sidebar.

```
1 call
    function Activity() {
20      var d = $("<div />");
21      for (var i = 0; i < 10; i++) {
22          $("<span />").appendTo(d).text(".");
23      }
54 calls
      setInterval(function () {
25      }, 300);
26      return d;
27  }
28
1 call
    $(document).ready(function () {
30      $(document.body).append(Activity());
31  });
```

**The call count which currently reads '54' increments by 1 every 300 milliseconds.**

*Verify that the timer clears appropriately*

Samantha adds a conditional statement that should cancel the timer when the activity indicator's DOM element is removed from the page. She verifies that it works correctly just by watching the call counts.

```
1 call
    function Activity() {
20      var d = $("<div />");
21      for (var i = 0; i < 10; i++) {
22          $("<span />").appendTo(d).text(".").css({ color: i =
23      }
10 calls
      var timer = setInterval(function () {
25          d.find("span:last").prependTo(d);
26          if (!$.contains(document.documentElement, d[0])) {
27              clearInterval(timer);
28          }
29      }, 300);
30      return d;
31  }
32
1 call
    $(document).ready(function () {
34      var indicator = Activity();
35      $(document.body).append(indicator);indicator
1 call
      setTimeout(function () {
37          indicator.remove();
38      }, 3000);
39  });
```

**After 3 seconds, the activity indicator was removed from the page and the timer's call count stopped at 10.**

## Summary

Throughout the implementation of this widget, Samantha was able to verify that the behavior of the code worked before she had added any code that would modify the page. The only time Samantha had to look at the web page was to verify its appearance. Having evidence of the program's execution in the code editor saved Samantha from needing to switch applications until there was actually something on the page to see. In this scenario, Samantha never had to interact with the debugger directly. No clicks or keystrokes were made solely for the sake of debugging.

## Retroactive Logging

In this scenario, Samantha is working on a Node.js program to count the total number of lines in all the files in a directory.

She starts with a sanity check. She verifies that she is using `fs.readdir`—the function for listing directory entries—correctly by calling it with an empty callback. When she runs the program, she sees that the callback is called, and by clicking on the call count, the log is populated with the values of the arguments to the function. Convention dictates that the first argument is an object encapsulating any error (so it being null is a good sign), and expanding the array allows her to see that the files she expects to find are there.

```
1   var fs = require('fs');
2
1 call
    function rwc(path, callback) {
1 call
5       fs.readdir(path, function () {
6       });
7   }
0 calls
    rwc('./test-dir', function (size) {
9       console.log('total size', size);
10  });
```

Log

(anonymous) (rwc.js:4) 1:02:49.547 **arguments[0]** = null **arguments[1]** = [Array:17] Backtrace →

**0** = ".gitignore"
**1** = "Gemfile"
**10** = "lib"
**11** = "log"
**12** = "public"
**13** = "script"
**14** = "test"
**15** = "tmp"
**16** = "vendor"
**2** = "Gemfile.lock"
**3** = "README.rdoc"
**4** = "Rakefile"
**5** = "app"
**6** = "config"

Samantha continues coding, running the program periodically to verify that the call counts make sense and that no exceptions are being thrown. She eventually notices a disparity: the `add` function, which gathers results, is being called fewer times (133) than the function that iterates over directory entries (140).

```
133 calls   var add = function (size) {
              totalSize += size;
              if (++count === totalCount) {
                  callback(totalSize);
              }
          };

48 calls  fs.readdir(path, function (err, filenames) {
              totalCount = filenames.length;
140 calls     filenames.forEach(function (filename) {
140 calls         fs.stat(path + '/' + filename, function (err, stats) {
                      if (stats.isDirectory()) {
                          rwc(path + '/' + filename, add);
                      } else if (stats.isFile()) {
                          add(stats.size);
                      }
                  });
              });
          });
```

She is faced with a difficult reachability question: find calls to the filename iterator callback that do not result in a call to the `add` function—either directly, or asynchronously.

Breakpoints don't work here because she cannot step into the `fs.stat` callback, and there would be 140 instances to step through besides. Naively adding log statements like `console.log("in iterator")` and `console.log("in add")` wouldn't work because it would be very difficult to find corresponding entries. Her solution is to use Theseus's log.

She clicks the call count next to the file entry iterator callback and the call count next to the `add` function. A log appears that looks like this:


Finding calls to the iterator callback that don't eventually call the `add` function can now be easily done by skimming the log, using the glyphs' shapes or colors, using the names of the functions, or using the shapes of the log entries themselves (`add` takes fewer arguments, so its lines are shorter).

The desired log entries happen to be clustered here:


They correspond to the empty directories in the `tmp` directory of a Rails project. When she adds a check for empty directories, the `add` function's call count jumps to 140. With a click she can see that the result looks reasonable:

```
1 call    rwc('./test-dir', function (size) {
              console.log('total size', size);
          });
```

**Log**

(anonymous) (rwc.js:31) 1:20:35.909 **size** = 295387 Backtrace →

console.log 1:20:35.909 "total size" 295387

## THESEUS IMPLEMENTATION

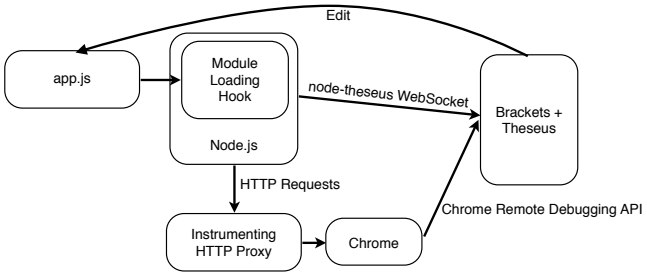


**Figure 2. System overview. node-theseus installs a module loading hook that instruments all JavaScript that gets added to the Node.js process. node-theseus then connects to Theseus via a WebSocket connection. When the user requests web pages, the requests are routed through a proxy server that instruments all JavaScript on the page. Theseus opens a WebSocket connection to the page to collect instrumentation data from there as well.**

Theseus consists of an extension for the Brackets editor, two modules that the extension uses to communicate with Chrome and Node.js, the JavaScript library called 'Fondue' that adds instrumentation hooks to JavaScript source code, and the trace-collecting module that gets injected into the programs being debugged. See Figure 2 for a graphical overview. The components are described briefly in this section.

### Fondue, the JavaScript instrumentation library

Fondue accepts JavaScript source code and returns JavaScript source code that behaves the same but also records all control flow information to a global trace object. Functions are changed to report the arguments received when entered, and the value returned when they exit.

Function expressions are rewritten so that the invocation at the top of the stack is recorded upon creation. That invocation is regarded as the function's asynchronous parent. When log entries exist for two invocations that would not otherwise

be connected in a call tree (because neither directly or indirectly called the other), if one is the asynchronous parent of the other, then they will be joined as parent and child with an 'async' flag added to the child's log entry.

Since objects in JavaScript are mutable, if the trace stored only a reference to function arguments or return values, their values might change by the time they were requested by the debugger. To protect against this, Fondue makes a shallow clone of all objects that it stores. Memory and accuracy can be balanced by adjusting the depth of the copy operation, which is currently set to 1 for objects and 2 for arrays (so that objects within arrays will be cloned).

#### Querying the program trace

When Fondue rewrites JavaScript to add the instrumentation hooks, it also prepends the definition of a global object to receive the trace data. That global object contains methods for accessing information such as the locations of function definitions in all loaded files, the number of times that functions have been called, and the log entries corresponding to a certain query. Theseus collects data by polling these functions at about 10 Hz over one of the connections described next.

#### Debugging server-side Node.js code

The user runs their code with the command `node-theseus app.js` instead of `node app.js`. node-theseus installs itself as a pre-processor of all JavaScript code that is loaded into that Node.js process. From then on, all code is instrumented with Fondue before being executed. node-theseus then opens a WebSocket server to listen for a debugging connection from Theseus.

#### Debugging client-side code in Chrome

The Theseus extension starts a web server that serves files from the user's project directory, processing any JavaScript it finds with the Fondue library. When the user opens a web page from that server, Theseus connects to that Chrome window using the Chrome Remote Debugging Protocol [4].

### EVALUATION 1: LAB STUDY

We conducted a lab study to answer several questions about how JavaScript programmers would use Theseus on a variety of JavaScript problems. Specifically, the study was designed to shed light on the five research questions below. The first three questions concern the ways in which we think Theseus would make programmers more efficient:

**RQ1.** How would programmers find correspondences between code and program behavior with Theseus?

**RQ2.** How would programmers use Theseus to find where chains of callbacks break down?

**RQ3.** Would programmers use Theseus's log to sort through tangled control flow problems?

#### Methodology

We recruited 7 participants, described in Table 1, to a 90-minute lab study. Subjects were required to have JavaScript programming experience. They were given 5 programming tasks: two 20-minute tasks and three 5-minute tasks. To facilitate within-subjects comparison, each participant was assigned to the Theseus or control condition for each task independently (but always with 2 tasks in one condition and 3 in the other). Subjects completed all of their control tasks first (using Chrome Developer Tools), then all of the Theseus tasks (during which Chrome Developer Tools were disallowed).

| Subject | Age | Gender | Prog. Ability | JS Ability | Uses JS |
|---|---|---|---|---|---|
| S1 | 24 | M | ••●•• | ••••● | Daily |
| S2 | 23 | M | ••●•• | •••●• | Daily |
| S3 | 20 | M | ••●•• | ••●•• | Few days/wk. |
| S4 | 29 | M | ••••● | •••●• | Few days/wk. |
| S5 | 24 | M | ••••● | •••●• | Few days/mo. |
| S6 | 21 | M | ••••● | ••●•• | Few days/mo. |
| S7 | 39 | M | ••••● | ••●•• | Not recently |

**Table 1. Programming Ability and JavaScript Ability are on a 5-point scale, with 1 labeled "Novice" and 5 labeled "Expert".**

The five tasks were as follows:

**A. Canvas Painter (20 minutes).** Subjects were given the source code for a browser-based drawing site with approximately 2,000 lines of JavaScript spread across 8 files. They were asked to fix the operation of the drawing tool.

**B. du (20 minutes).** Subjects were given the skeleton for a Node.js command-line tool for calculating the total size of all files in a directory and asked to complete it.

**C. Laggy AJAX UI (5 minutes).** Subjects were given a web page with 25 lines of JavaScript for a web page that downloaded JSON from the server and displayed it in a popup. Subjects were asked to determine why it took so long for the popup to appear. The solution was a delay hard-coded into the server.

**D. Faulty Auto-Complete (5 minutes).** Subjects were given the code for both the server (80 lines of Node.js) and client (63-line HTML file with embedded JavaScript) of a web page that showed auto-complete search results from an address book. Subjects were asked why the results never displayed. The problem was a logic error on the client while processing the results.

**E. Real-Time Chat (5 minutes).** Subjects were given the code for the server (33 lines of Node.js) and client (31 lines of JavaScript on a web page) of a real-time chat site. Subjects were asked why messages from one window did not appear in the other. The problem was that the message name on the client and server was mis-matched.

Task E was selected for RQ2 because it involved broken callback chains. Task B exemplified tangled asynchronous control flow. All of the tasks except B required the user to discover correspondences between code and behavior on the page. Because the code was broken in some way in tasks A, D, and E, users were forced to validate many assumptions they made about the code.

| Subj. | A | B | C | D | E | Ease of Use | Would Use | Would Recom. |
|---|---|---|---|---|---|---|---|---|
| S2 | • | • | ✓ | ✓ | ✓ | ••••●• | ●••••• | ●••••• |
| S7 | • | • | ✓ | • | ✓ | ••●••• | ••●••• | •••••● |
| S5 | • | • | ✓ | ✓ | ✓ | ••••●• | ••●••• | •••••● |
| S1 | • | • | ✓ | ✓ | • | ••••●• | ••••●• | ••••●• |
| S6 | • | • | ✓ | • | • | •••••● | •••••● | ••••●• |
| S3 | ✓ | ? | ✓ | • | ✓ | ••••●• | •••••● | •••••● |
| S4 | ✓ | ✓ | ✓ | ✓ | ✓ | ••••●• | •••••● | •••••● |

**Table 2. Summary of study results. The cells in the columns labeled A–E contain ✓ if the subject successfully completed that task. The cells are shaded blue if the task was done with Theseus. The correctness of S3's solution for task B was unclear, so that cell contains a question mark. Ease of Use, Would Use, and Would Recommend are on a 5-point scale, with 1 lowest and 5 highest.**

At the conclusion of the study, we verbally asked five questions regarding their opinion of Theseus. The results will be presented in the next section.

### Results

The results of the study are summarized in Table 2. Because of the small number of participants, we were unable to establish any statistically significant relationships between participants' success rates and the tools they used. A chi-squared test found no relationship between using Theseus and the participant's ability to complete the tasks successfully ($\chi^2(1, N = 34) = .119, p = 0.73$).

*RQ1: How might programmers find correspondences between code and program behavior with Theseus?*

Participants frequently sought code correspondences using Theseus by keeping the call counts and code coloring on the screen as they interacted with the program they were working on. They were pleased with how much information they could absorb this way, an experience S1 described like this: "[Theseus] feels really interactive. [As opposed to breakpoints], it's more of a 'watch and see what happens' thing, which I like." This behavior was not observed during Tasks B or E, likely because Task B involved writing a non-interactive command-line tool and the problem in Task E resided in the logic of a single function. However, during Task A, S1 and S3 used this strategy many times (6 times and 3 times, respectively) during the task. Four of the six participants used this strategy during Tasks C and D. The only participants who did not use this strategy on Tasks C and D were S2 and S4, likely because they spent their first 20 minutes with Theseus working on Task B, the non-interactive command-line tool.

There was some disagreement about whether the call counts were useful for finding correspondences when the user had no idea where to begin. S5 said, "how the call counts changed live when I interacted with the application ... was especially useful for Canvas Painter because it was a lot of source code and I didn't really know where to start." As a result, S5's rating of whether they would use Theseus outside the study depended on the size of the project: rated 4 out of 5 if the code base is large, but only 1 out of 5 if the code base is small. S1 had the opposite opinion, stating, "I felt like it was the least useful when I wasn't sure where the problem was. So in the canvas thing, I didn't know where the issue was, and there's not much of a global scope with Theseus. ... When I didn't know where to start, there was no way to find a global call stack and identify candidate starting points. ... [In short, Theseus is] more useful on a narrow scope, less useful on a global scope."

Participants were interested in the time at which the call counts changed if they were interacting with their application, but also the total number. A changing call count could alert the programmer to surprising or revealing information, such as when S3 watched the call counts during Task A. At one point, S3 thought aloud, "I get 2 mouse up actions [every time I click]. Huh." Then while watching the call counts and clicking a second time, they exclaimed, "Aha!" as the nature of the problem became more clear. S5 noted that they had become fixated on a handful of functions while trying to narrow down the location of some strange behavior because "it seems weird to me that I get 2 mouse ups every time I click, while I only get 1 mouse down. ... I'd expect the call counts to be the same for both of them, but they're not." S4 and S6 also used the fact that a function was called 17 times as verification that it was being called once for each file in the directory during Task B, since they had checked that there were 17 files.

The call counts also turned out to be useful for verifying that a code change had had the desired effect. In S4's case, the fact that their change caused a function to be called a different number of times was encouraging. The call count seemed a reliable enough indicator for checking that the new behavior was correct that they performed no further tests.

*RQ2: How might programmers use Theseus to find where chains of callbacks break down?*

Finding where chains of callbacks break down is an important subtask of finding code correspondences. In JavaScript, functions typically cannot block while waiting for the result of an I/O operation, which forces the programmer to split computations into multiple callback functions, with no guarantee that control will flow successfully from one function to the other.

We noted several points during the study when participants using Theseus were able to quickly, sometimes immediately, locate the cause of a broken call chain, using the code coloring and call counts. In one instance, S4 opened a source file and was immediately drawn to a network event handler that had never been called, becoming suspicious because it looked like a handler that should have fired several times if the page had been working correctly. This was in contrast to S3's experience using a breakpoint debugger, in which they set breakpoints and reloaded the page three times before they finally determined how much of the code had actually executed.

*RQ3: Would programmers use Theseus's structured log to sort through tangled control-flow problems?*

S4 named the log as Theseus's most useful feature, saying that Theseus is most useful "if you have recursion problems," referring to Task B in which his solution involved recursive asynchronous operations. S3 dubbed the pills "automatic silent breakpoints." S1 compared the log to typical log output,

saying, "[Theseus] is a lot more focused ... with console.logs it's global. ... [With Theseus] you can pick the scope you want to look at on the fly." S4 summarized his opinion of the log like this:

> "It gives you what you would do if you were really careful and did console.log every function. Yeah, so I didn't have to console.log. This saves at least one or two iterations if the first thing you log is really the thing you need. If you need to go through and look more, then this can save a lot more iterations. ... This should be in Chrome. ... This should be in every JavaScript debugger. This is very useful."

S4 would often click the pills for several functions at once, saying, "all the time, the thing that I wanted to do first is select all the functions and then see the whole tree." Showing the asynchronous call tree for all the functions of interest in the file helped him to locate the points of interest. He cited the lack of a 'Select All Pills in File' command as the reason he rated Theseus' ease of use as 4/5 instead of 5/5. S6 felt similarly, at one point saying aloud, "These are the four functions that are interacting," and without pausing, enabling the pills for those four functions to see how they related.

## EVALUATION 2: INTERVIEWS

A limitation of the first study, like most lab studies, was its short duration and experimenter-chosen tasks. Programmers had little time to change their programming behavior in light of a completely new tool, and no chance to use it on their own tasks. To better understand how Theseus's always-on visualizations might work in practice, we ran a second study that gave Theseus to programmers for about a week, then interviewed them to see how their programming behaviors had adapted to the tool.

### Methodology

We recruited a team of nine professional JavaScript programmers who work on a code base of about 80,000 lines to participate in a week-long study. The particiapnts were all male. Because they were professional programmers, the subjects in this study were significantly more experienced than the university students in the previous study. Subjects received $25 as compensation.

We introduced Theseus to the study participants at a group meeting, and encouraged them to use Theseus for one week. We asked them to document a programming problem in the next week for which Theseus was or was not useful, and to save a snapshot of the code at that point in time. At the end of the week, we conducted hour-long semi-structured interviews with each participant.

Each interview was comprised of two parts: First, we asked the participant to walk through the problem they documented, as well as their solution. Second, we asked partitipants to complete the Canvas Painter programming task from the previous study. Subjects were instructed to use whatever programming tools they were most comfortable with. However, if they did not choose to use Theseus on their own, we asked them near the end to continue working with Theseus instead of their preferred tool.

| Subject | Theseus Use Pre-Interview (self-reported) |
|---|---|
| S20 | 4 hours |
| S21 | 3 hours |
| S22 | 1 hour |
| S23 | 0.25 hours |
| S24 | 2–4 hours |
| S25 | <1.5 hours |
| S26 | <0.25 hours |
| S27 | <0.5 hours |
| S28 | 1.5–2 hours |

**Table 3. Each subject's self-reported time spent using Theseus before the interview.**

In this study, we evaluated three hypotheses about how always-on visualizations might aid programmers in debugging tasks:

**H1.** People will verify code behavior by running it instead of reading and guessing.

**H2.** People will notice errors or oddities for investigation.

**H3.** People will locate the code responsible for a particular behavior more accurately/confidently.

Additionally, we evaluated two hypotheses regarding the overall perception of always-on displays:

**H4.** People will feel like they waste less time interacting with debugging tools with always-on displays active.

**H5.** People will want more always-on displays.

Finally, we were interested in gaining a qualitative understanding of *how* always-on displays fit into a programmer's workflow, namely 1.) the situations when they were found to be appropriate, helpful, or preferred, and 2.) the down-sides that programmers found, such as distraction and information overload.

A prompted think-aloud protocol was used, with the interviewer's prompts guided by the hypotheses. That is, when a subject seemed to be performing a relevant act, such as reading code, the interviewer would elicit the participant's justification for his current actions. We made a screen and audio recording of each interview. The audio was transcribed by one researcher. Then, the transcripts were coded for relevance to the above hypohtheses by five researchers (one of whom was the interviewer).

### Results

Subjects spent varying amounts of time using Theseus during the week between its introduction and their interview. The times are summarized in Table 3.

The evidence related to each of the hypotheses is presented under each of the headings below.

*H1: People will verify code behavior by running it instead of reading and guessing*

Subjects did not activate a debugger until they got stuck. Upon starting the Canvas Painter task, all subjects began by skimming the source code without the aid of a debugger. The task was to change the way the page reacted to mouse events (specifically the end of a drag), so subjects gravitated toward mouse-related code. S21 started by searching for 'click' but the rest just skimmed files that had suggestive names for code that looked like it contained mouse event handlers. 4 of the 9 subjects said something like "familiarize myself with where all the code is" for this step (S26's words). S22 went so far as to say, "I try to stay out of the debugger as much as possible because it's a time suck."

Three subjects eventually ran the code as part of the initial code location step. S23 used Chrome's event breakpoints to have the step debugger stop every time JavaScript code ran in response to a click event. S24 and S27 ran Theseus to tell which of several mouse handlers was the one he should be interested in. S27 explained that if he didn't have Theseus, he would probably spend more time just reading the code, since breakpoints are tricky to use with mouse events (because using the debugger requires using the mouse) and that adding log statements is tedious.

#### *H2. People will notice errors or oddities for investigation*

Code coloring affected the reading process of the programmers. S24 was drawn to a particular section of unexecuted code and read it to figure out whether it should have executed, according to how he thought the code worked. S25 skipped reading portions of a file, saying, "Okay, nothing called in this." However, his attention was drawn to code that had been called even if it was not related to the code he was attempting to locate. S20 ignored the coloring and read code that Theseus marked as unexecuted to understand the behavior of the page, despite knowing that unexecuted code could not have had any effect.

During an explanation, S21 described how a certain situation could not occur because a certain function wasn't being called. He checked the source and discovered that actually the function *was* being called, which led to further exploration and a revised explanation.

Call counts were effective in revealing oddities or confirming correct behavior. S20, S21, S24, and S25 encountered instances where the absolute number of calls to a function drew their attention to a problem or an aspect of the code that they did not yet understand. S20 noticed a call count while forming a hypothesis about the code and used it to inform his thinking, saying, "Hmm, this is only called one time ... does that mean the ... clearing the drawing is not another drawing ..." S21 had his attention drawn to call counts that were surprisingly high, which allowed him to make a guess at what the code was for, saying "This was called a bunch, 319 times... maybe they're simulating dragging." S24 used Theseus on a project that loaded 100 images asynchronously and noted that "`img.onload` is stopping at 100. That's good, that's perfect."

The differences between call counts also proved important. S21 noticed that a callback was occurring twice every time he performed an action on the page, instead of only once as he had expected. Both S24 and S27 found bugs in their respective applications when they noticed that a pair of functions that should have been called the same number of times actually had different call counts. In S24's case it was the start and end of a callback chain, and for S27 it was a pair of mouse-down and mouse-up handlers.

The colors of the call counts were occasionally useful as well. S28 noticed the red coloring of a call count that indicated an exception in the function he was reading, and stated that he had not noticed the exception when it was printed to the console.

#### *H3. People will locate the code responsible for a particular behavior more accurately & confidently*

Subjects used total call counts, changes in call counts, and lack of calls to make informed guesses about what code was for. S21, S23, and S24 used the side-by-side technique to verify that they understood what the code they were looking at was responsible for the behavior they thought it was. S21 was satisfied with weak evidence in one case, saying, "So this was called 7 times. ... Seems about right. I didn't draw that many things." In the process of watching call counts change for one function, S21 noticed that the mouse-up handler nearby was being called more times than expected. S23 used the *lack* of changes to a call count to determine that they were incorrect about which handler was used for a particular action. When reading a screen full of potential handlers for an event, S20 used the code coloring to decide which was the one he was looking for—it was the only one that wasn't gray.

S26 adapted a trace comparison strategy he was used to using with log statements to the information available in Theseus. S26 wished to compare what happened in the code when he triggered the bug with what happened when he performed the same task in a way that did not trigger the bug. To do so, he reset the call counts by reloading, performed the task in one way, observed the counts, then repeated those steps with the alternate means. The process was tedious, but the difference in the call counts gave him an idea of where to look.

S20 and S25 found that Theseus would have been more helpful with locate responsible code if it visualized activity at the project level.

#### *H4. People will feel like they waste less time with always-on tools*

The interviews revealed relatively little evidence about perceived or actual time-wasting. S20 observed that Theseus's ability to associate each callback with the invocation that created it was very hard to accomplish with other debuggers, and perceived it as a time-saver. S23 felt that the "biggest JavaScript debugging time suck" was determining why an event handler wasn't being called, and Theseus answered that question more efficiently than inserting log statements by hand. But S23 and S25 both spent time waiting while Theseus rendered a large group of log statements.

#### *H5. People will want more live displays*

4 subjects (S22, S25, S27, and S28) expressed interest in more always-on displays, usually as extensions to Theseus's current interface. S22 wished for Theseus to show the time spent in every function, S22 and S25 wanted the file-level counterpart to the function-level call counts, and S27 and S27 wanted information about the state changes on individual lines.

Subjects S20 and S21 preferred step debuggers to Theseus. S20 felt strongly that breakpoint debuggers were more natural. S21 switched away from Theseus to Chrome's step debugger, saying "I can't quite see ... what its current state is and how that influences which conditional we're going down," and that the "[step debugger] mental model is easier to understand." S21 pointed out that with a step debugger, he could also watch the web page update as he stepped.

*When do programmers find Theseus to be helpful?*

Breakpoints interfered with reproducing bugs related to mouse gestures, but Theseus did not. S23 and S24 chose to use Theseus instead of breakpoints because of the way breakpoints would affect the behavior of the program. In S23's case, he feared that interacting with the step debugger would interfere with reproducing bugs on the web page with the mouse. S24's code involved downloading many files in parallel and the code that executed in the meantime, so stopping at breakpoints would affect the order that events occurred. S23 said that they had considered adding log statements as well, but that it was more tedious than using Theseus's log.

S20 chose to use breakpoints at first, but eventually switched to Theseus. S20 started by setting a breakpoint, then running the code but failing to hit the breakpoint. He repeated those steps 2 more times before saying, "I can see that this kind of thing might be a lot easier to see ... with Theseus." He activated Theseus and was able to see which functions were actually used during his mouse gesture.

However, S21 preferred using breakpoints to using Theseus. S21 thought that omniscience was the useful aspect of Theseus, saying that if his browser's breakpoint debugger were omniscient, he would use that instead of Theseus's log. He pointed out that breakpoints allowed him to view the entire state (including the contents of the web page) after every step.

S27 did not use Theseus because there would have been "a bunch of calls all over the place" and he feared that he wouldn't be able to see which were related to his bug.

S28 discovered that Theseus introduced timing delays in his multi-threaded animations which changed the characteristic of the bug he was investigating.

*Downsides of always-on displays*

Two subjects experience information overload when interacting with the log. S23 said that showing all of the arguments and return values simultaneously made him unable to parse any of it. S25 simply noted that there was a lot of information which he did not need. On the other hand, S22 appreciated that the log showed a lot of data at once.

## CONCLUSIONS AND FUTURE WORK

This paper presented Theseus, an IDE extension that visualizes run-time behavior within a JavaScript code editor. Theseus visualizes the program's run-time state using code coloring for unreached functions, call counts for executed functions, and automatic retroactive logging of parameters and return values. A lab user study and a field deployment found that programmers enjoyed the availability of reachability coloring and call counts, and adopted new problem-solving strategies to take advantage of their strengths.

Future work in always-on programming visualizations should take two directions. First is increasing the reach of Theseus's omniscience, by capturing state at finer-grained locations than function entry and exit points, and information about heap data structures and user interface (DOM) state. A key challenge in this area is capturing information without adversely affecting performance, so that the logging can remain always-on. Recent work on replay debugging [2] will help, but performance still needs to be improved. The second research direction is studying new kinds of runtime information to visualize, such as variable types, uses of undefined variables and fields, and exception-throwing expressions, and new designs for displaying them unobtrusively with the code.

## ACKNOWLEDGMENTS

This work was supported in part by Adobe, and by the National Science Foundation under award number SOCS-1111124. Any opinions, findings, conclusions, or recommendations in this thesis are the authors', and they do not necessarily reflect the views of the sponsors.

## REFERENCES

1. Arends, R. traceGL, Sept. 2013. `https://trace.gl/`.
2. Burg, B. Timelapse: Interactive Record/Replay for the Web. `http://homes.cs.washington.edu/~burg/projects/timelapse/research.html`.
3. Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM) 22*, 2 (2013), 14.
4. Google. Remote debugging protocol - Chrome DevTools, Sept. 2013. `https://developers.google.com/chrome-developer-tools/docs/debugger-protocol`.
5. Gould, J. D. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies 7*, 2 (1975), 151–182.
6. Griffiths, D. Scheme Bricks, Sept. 2013. `http://www.pawfal.org/dave/index.cgi?Projects/Scheme%20Bricks`.
7. Kato, J., Mcdirmid, S., and Cao, X. DejaVu: Integrated Support for Developing Interactive Camera-Based Programs. In *Proc. UIST 2012* (2012).
8. Ko, A. J., and Myers, B. A. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proc. SIGCHI 2004*, vol. 6 (2004).

9. LaToza, T. D., and Myers, B. A. Developers Ask Reachability Questions. In *Proc. ICSE 2010*, vol. 1, ACM Press (New York, New York, USA, 2010).

10. LaToza, T. D., and Myers, B. A. Visualizing Call Graphs. In *Proc. VL/HCC 2011*, Ieee (Sept. 2011).

11. Lieberman, H., and Fry, C. Bridging the Gulf Between Code and Behavior in Programming. In *Proc. SIGCHI 1995*, CHI '95, ACM Press/Addison-Wesley Publishing Co. (New York, NY, USA, 1995).

12. Meier, M. S., Miller, K. L., and Pazel, D. P. Experiences with Building Distributed Debuggers. In *Proc. SIGMETRICS 1996* (1996).

13. Microsoft. Debug Your App by Recording Code Execution with IntelliTrace. `http://msdn.microsoft.com/en-us/library/vstudio/dd264915.aspx`.

14. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al. Scratch: programming for all. *Communications of the ACM 52*, 11 (2009), 60–67.

15. Schrock, E. Debugging AJAX in Production. *ACM Queue* (2009).

16. Sillito, J., Murphy, G. C., and De Volder, K. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, ACM (New York, NY, USA, 2006), 23–34.

17. Victor, B. Learnable Programming. `http://worrydream.com/#!/LearnableProgramming`, 2012.