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 [4, 6, 8]. 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 [8, 18]. Unfortunately, code behavior is invisible in most programming environments because 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 by displaying information about execution sooner.

Our solution is a code editor extension called Theseus (Figure 1). Theseus visualizes the program's run-time state using 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. A program trace is collected so that clicking on one of those call counts can immediately add entries to a log showing the arguments and return values of every invocation. 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 [8].

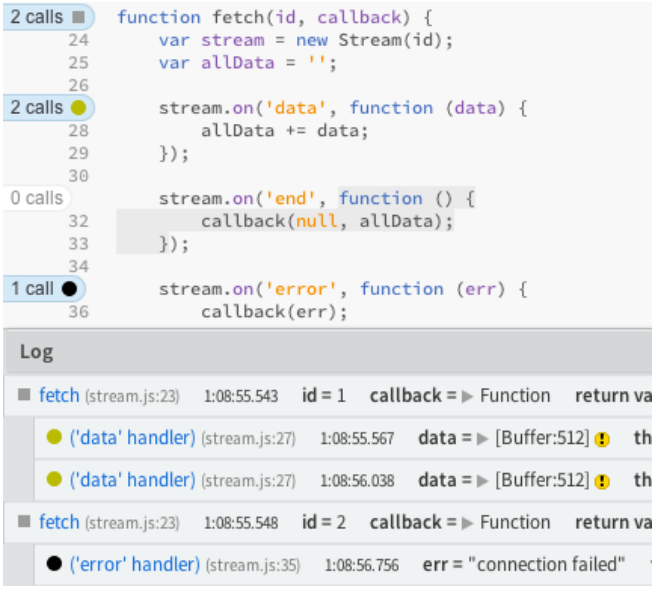


Figure 1. Theseus shows call counts for every function, and an asynchronous call tree allows the user to see how functions interact. In the log below the code, users can see which call to `fetch` corresponds to the failure without adding any debugging-specific code or re-executing their program.

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 without Theseus. Theseus users identified the locations where chains of callbacks broke down more quickly (often

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

CHI 2014, April 26–May 1, 2014, Toronto, ON, Canada.

ACM 978-1-4503-2473-1/14/04$15.00.
http://dx.doi.org/10.1145/2556288.2557409

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 open-source debugging interface called Theseus that answers reachability questions in the context of the programmer's source code and exemplifies the always-on design principle
- 2 studies of how Theseus affects programmers' behavior
- An open-source library called Fondue for collecting JavaScript traces in real-time for always-on interfaces

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

RELATED WORK

A live programming interface provides feedback to the programmer after every change about the new program's behavior [20]. Many interface ideas have arisen from live programming research, such as McDirmid's probes and traces interface [12], Swift's live code annotations for temporal recursion [19], and Bret Victor's demonstrations of a programming environment with domain-specific visualizations [21]. Theseus shares aspects with all three of these interfaces, but is designed to work with JavaScript in non-live environments. Block-based programming environments like Scratch and Scheme Bricks highlight code as it is executed, which can help users draw correspondences between the code and program behavior, though that information does not persist, like Theseus's coloring and call counts [15, 5]. Theseus's coloring plays a similar role to continuous testing interfaces, which provide the programmer feedback on their progress by indicating which tests are passing and failing [16].

Reacher [9] 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 visualizes execution that has 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's log and Reacher provide complementary views of the same kind of information and it would be fruitful to combine them.

Whyline [7] is a debugging interface that users can ask questions like, "Why is this widget blue?" Whyline answers this question by generating a slice of the program containing the UI events and branches that led to the widget being blue. Since Theseus visualizes the execution of individual functions (which is usually completely invisible), it opens up a world of information about which programmers could ask questions with an interrogative debugger such as Whyline.

The idea of adding edges to the call graph (such as those Theseus adds for asynchronous calls) to aid debugging goes back to ZStep [10], 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 [14] 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 [17], and more generally across client-server boundaries [13].

Timelapse offers self-contained, fully-replayable traces of entire web pages [1]. Timelapse integrates tightly with WebKit browsers in order to apply low-level algorithms based on virtual machine record/replay. Theseus's instrumentation library, Fondue, is based on source rewriting. The overhead of collecting the trace is much worse than that of Timelapse, but Fondue is flexible and can be adapted to new contexts—such as instrumenting Node.js or creating an instrumenting proxy server—more easily as a result.

INTERFACE DESIGN

Theseus is designed to address three types of challenges that programmers face: 1) cost of information: the inefficiency of querying a debugger about program execution, 2) finding code/behavior correspondence: identifying which code is responsible for a particular program behavior, and 3) understanding asynchronous control flow.

These areas of focus come from research into the types of questions programmers ask while programming [8, 18], primarily the first three categories of questions identified by Sillito et al. (finding initial focus points, building on those points, and understanding a subgraph) and reachability questions over a call graph with asynchronous links.

In this section, we demonstrate how Theseus's always-on visualizations aid the programmer with these challenges by describing two sessions with Theseus.

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 constructor for the widget, and a callback function to instantiate one when the page loads. After writing only the definitions of those functions, Samantha opens the page in a browser.

As shown below, the way her code is displayed immediately changes in two ways: code that has not executed is given a gray background, and the number of times each function is called is displayed in the margin to the left of each function. From these cues, Samantha infers that her syntax was correct, that she registered her document-ready event handler correctly, and that no exception was thrown because none of the call counts are colored red.

```
0 calls   function Activity() {
      5   }
      6
1 call    $(document).ready(function () {
      8   });
```

The Brackets 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.

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() {
      5       var d = $("<div />");
      6       for (var i = 0; i < 10; i++) {
      7           $("<span />").text(".").appendTo(d);
      8       }
56 calls      setInterval(function () {
     10       }, 300);
     11       return d;
     12   }
```

The call count which currently reads '56' increments by 1 every 300 milliseconds, making it easy to pick out code responsible for the animation.

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: when the timer stops, the call count for its handler will stop incrementing.

```
1 call    function Activity() {
      5       var d = $("<div />");
      6       for (var i = 0; i < 10; i++) {
      7           $("<span />").text(".").appendTo(d);
      8       }
10 calls      var timer = setInterval(function () {
     10           d.find("span:last").prependTo(d);
     11           if (!$.contains(document, d[0])) {
     12               clearInterval(timer);
     13           }
     14       }, 300);
     15       return d;
     16   }
     17
1 call    $(document).ready(function () {
     19       var indicator = Activity();
     20       $(document.body).append(indicator);
1 call        setTimeout(function () {
     22           indicator.remove();
     23       }, 3000);
     24   });
```

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

Discussion

Throughout the implementation of the widget, Samantha was able to verify the behavior of her code even before it had any output. The only time Samantha had to look at the web page was to verify its appearance. Samantha never had to interact with the debugger directly. No clicks or keystrokes were made solely for the sake of debugging. This greatly reduced the cost of the information she received, and made it easy to recognize that the code responsible for the timer was behaving as expected.

Retroactive Logging

In this scenario, Samantha is working on a Node.js program to count the total size of 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 an error if there is one (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 call    function du(path, callback) {
1 call        fs.readdir(path, function () {
      7       });
      8   }
      9
0 calls   du('./test-dir', function (size) {
     11       console.log('total size', size);
     12   });
```

Log

('readdir' callback) (du.js:4)	10:55:31.320	arguments[0] = null	arguments[1] = [Array:16] 0 = ".gitignore" 1 = "Gemfile" 2 = "Gemfile.lock" 3 = "README.rdoc" 4 = "Rakefile"

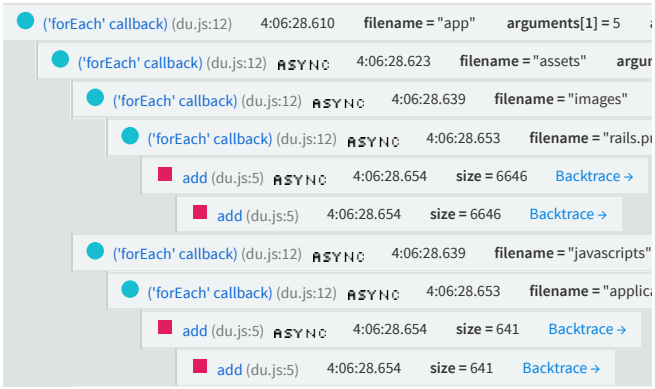
Samantha continues coding, running the program periodically to verify that the call counts make sense and that no exceptions are being thrown. She eventually notices a disparity: the `add` function, which gathers results, is being called fewer times (84) than the function that iterates over directory entries (91).

```
84 calls    var add = function (size) {
      6       totalSize += size;
      7       if (++count === totalCount)
      8         callback(totalSize);
      9     };
42 calls    fs.readdir(path, function (err, filenames) {
     11       totalCount = filenames.length;
91 calls      filenames.forEach(function (filename) {
     13         var childPath = path + '/' + filename;
91 calls        fs.stat(childPath, function (err, stats) {
     15           if (stats.isDirectory()) du(childPath, add);
     16           else if (stats.isFile()) add(stats.size);
     17         });
     18       });
     19   });
```

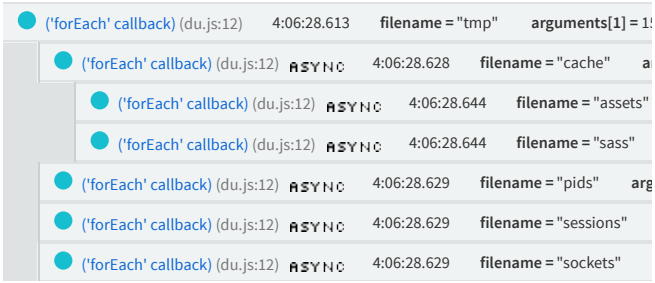
She is faced with a difficult reachability question: find calls to the filename iterator callback that do not result in a call to the `add` function—either directly, or asynchronously.

Breakpoints don't work here because she cannot step into the `fs.stat` callback, and anyway there would be 91 instances to step through. 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 done by skimming the log, using the glyphs' shapes or colors, the names of the functions, or 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 sub-directories of the `tmp` directory of a Rails project. When she adds a check for empty directories, the `add` function's call count jumps to 140. She clicks the `du` callback function's call count to see the final answer.

Discussion

Samantha answered several more common programming questions (what are the arguments to this function? what are the values of the arguments at run-time? how is control (not) getting from here to here?), including a difficult-to-answer reachability question involving recursive asynchronous function calls.

In addition to answering questions, the always-on features of the interface provide information scent that Samantha uses for sanity checks and identifying a silent bug that would not have been otherwise caught without a test suite [3]. Though the information is always present, the ambient display heuristics of Mankoff, et al. do not indicate any obvious deficiencies of Theseus's always-on features and, as described in the coming evaluation section, programmers find a variety of ways to use that information [11].

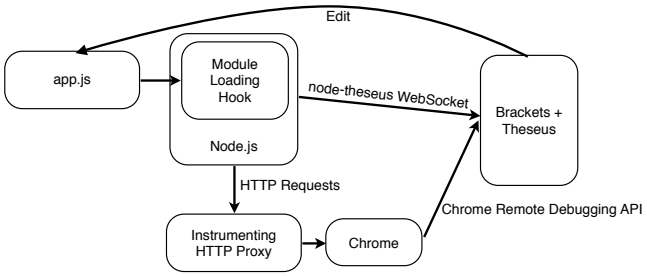


Figure 2. System overview. To instrument Node.js code, node-theseus hooks into the Node.js module loader to pre-process files with Fondue. To instrument code in the browser, HTTP requests are redirected to a proxy server. To read the Node.js trace and execute queries, node-theseus connects to Theseus via a WebSocket. Browser traces are read by connecting to Chrome using Chrome's Remote Debugging Protocol.

IMPLEMENTATION

Theseus consists of an extension for the Brackets editor, two modules that the extension uses to communicate with Chrome and Node.js, the JavaScript instrumentation library called "Fondue". See Figure 2 for a graphical overview. The components are described briefly in this section, and their complete source code is available online[2].

Fondue, the JavaScript instrumentation library

Fondue transforms JavaScript source code to record a trace of its execution to a global trace object. Functions are changed to report the arguments they receive, the values they return, and other information. This information is used to build a call graph as the code executes.

The call graph that Fondue builds also contains edges for asynchronous function calls. For example, in the code below, `foo` schedules `bar` to be executed 1,000 milliseconds later. There is no direct call chain connecting `foo` and `bar`, which is why a programmer could not step from `foo` into `bar` in most JavaScript debuggers.

```
function foo() {
    setTimeout(function bar() {}, 1000);
}
```

Fondue creates edges from an invocation of `foo` to every invocation of `bar` that results from it. To make this work, Fondue alters `bar`'s function definition so that when it is evaluated, the invocation at the top of the stack (in this case, `foo`) is recorded in the closure. That invocation is regarded as the function's asynchronous caller. Then, when no synchronous call chain connects `bar` to another function in the log, asynchronous call chains will be used instead if they exist. An

[2]Theseus: **`https://github.com/adobe-research/theseus`**
Fondue: **`https://github.com/adobe-research/fondue`**

'async' flag is displayed in the child entry in the log when this happens.

Mutability

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 use 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 given query. The functions are designed to be polled so that the consumer can control the refresh rate, which for Theseus is about 10 Hz. The simple JSON API and polling model make it easy to use Fondue in various contexts, such as those described in the next two sections.

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. The WebSocket server simply applies the deserialized messages as arguments to functions on the trace object, then sends the return value back as JSON.

Debugging client-side code in Chrome

Theseus starts a web server that serves files from the user's project directory, using Fondue to process any JavaScript embedded in HTML or served as `.js` files. When the user opens a web page from that server, Theseus connects to that Chrome window using the Chrome Remote Debugging Protocol, which supports evaluation of arbitrary JavaScript expressions on the page and receiving the return values as JSON[3].

EVALUATION 1: LAB STUDY

We designed the first study to determine whether an effect on programmers' behavior using our interface could be discerned, and to observe how well programmers could use our always-on interface and log tool in a handful of typical programming scenarios. We anticipated that the call counts and reachability code coloring would allow users to find correspondences between code and program behavior quickly, and would allow users to tell at a glance where chains of callbacks were breaking down.

[3] `https://developers.google.com/chrome-developer-tools/docs/debugger-protocol`

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. Participants of Evaluation 1. Programming Ability and JavaScript Ability are on a 5-point scale, with 1 on the left labeled "Novice", and 5 and the right labeled "Expert".

In this study, we focused on three research questions regarding three types of programming tasks:

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). They performed the tasks on a computer we provided while their screen was recorded and an observer took notes. Subjects completed all of their control tasks first using CDT (Chrome Developer Tools), then all of the Theseus tasks (during which CDT were disallowed). Although Theseus's functionality could not entirely replace CDT, we expected that programmers would use it exclusively if under time pressure and given the choice. Thus, we disallowed CDT in the experimental condition to maximize the available time to observe subjects using Theseus.

We put together five programming tasks which would allow us to test our hypotheses two of which were longer problems with more involved solutions, and three short tasks from which we had hoped to gather timing data:

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 a bug where a line-painting operation worked if the user clicked the start and end points, but not if they dragged the mouse.

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. They were asked to use only the asynchronous filesystem API calls.

C. Laggy AJAX UI (5 minutes). Subjects were given a web page with 25 lines of JavaScript for a web page that down-

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 the lowest agreement rating of 1 on the left end and the highest agreement rating of 5 on the right end.

loaded 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.

The source code for all tasks is available online[4].

All of the tasks except B required the user to discover correspondences between code and behavior on a web 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. Task B was chosen because it exemplified tangled asynchronous control flow. Task E was selected because it involved broken callback chains.

Each subject was given the At the conclusion of the study, we verbally asked five questions regarding their opinion of Theseus.

Results

The results of the study are summarized in Table 2. We found no 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." 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?

[4] `https://github.com/alltom/theseus-lab-study-files`

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 call counts "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."

S4 would often click the call counts 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 [call counts] 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 call counts for those four functions to see how they related.

Discussion

Users seemed to integrate the call counts and code coloring displays into their programming flow as was demonstrated by the widespread use of the side-by-side strategy. We did not see any clear evidence of whether call counts can help find code correspondences in a large, unknown code base, but users used them successfully in several different ways when the search space was small. We learned that the total number of calls, the differences in the total over time, and the relationships between call counts of different functions are all relevant to programmers, and observed situations in which each was used. We noted that participants had to memorize call counts that were separated by time or space, which software could have helped with. Participants also desired insight into activity throughout the project and not just the code that was visible on the screen.

Participants responded favorably to being able to add entries to the log without restarting their program, and used the log's call tree structure to see how several functions interact. The fact that asynchronous edges were used to build the call tree did not play a large role in participant's experiences.

EVALUATION 2: INTERVIEWS

Programmers in the last study seemed to perform about as well with Theseus as they did without, so we decided to look more closely at the behaviors we expected would contribute to their greater success when using Theseus, primarily usage of the increased information scent in the code, from which we expected an increased ability to draw code correspondences. We were also interested in whether programmers would adopt Theseus if they had more time to become better at using it.

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 participants 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.

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

H1. People will notice errors or oddities for investigation.

H2. People will locate the code responsible for a particular behavior more accurately/confidently.

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

H3. People will verify code behavior by running it, to take advantage of always-on displays that display runtime information.

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.

Theseus Use Pre-Interview (self-reported)	Participants
<15 minutes	S23, S26
<1.5 hours	S22, S25, S27
>3 hours	S20, S21, S24, S28

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

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

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 hypotheses by five researchers (one of whom was the interviewer).

The observer's notes from the interviews were broken into 1,611 fragments of about one sentence each. The fragments were loaded into Frenzy [2], a crowd work tool that the coders used to tag each fragment with the research questions to which it was relevant (if any). Each coder voted those tags up or down in the case that the fragment had already been categorized. The results sections were written by summarizing those categorized piles. Since coders did not work independently, inter-rater reliability was not measured.

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 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.

H2. 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.

H3: People will verify code behavior by running it instead of reading and guessing

Subjects did not activate a debugger or Theseus until they got stuck. Upon starting the Canvas Painter task, all subjects began by skimming the source code without the aid of any always-on displays. 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 they 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 interrupt streams of mouse events, and that adding log statements is tedious.

H4. People will feel 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. However, S23 and S25 had to wait for Theseus to render a large group of log statements once each.

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 S28 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 ... 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.

Downsides of always-on displays

Two subjects experienced 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.

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.

Discussion

The programmers in this study seemed to think of Theseus as a debugger and not a tool to be left running all the time. Their feedback suggested that the overhead of instrumenting their entire project was too great, which may have been what kept in the as-needed category of tool. This would explain why only two programmers used Theseus to verify their initial focus points were correct.

Again, we observed users using live-updating call counts in the same variety of ways as in the lab study, although we also saw evidence that presentation may need to be improved to reduce errors such as S20 reading unexecuted code to find the source of a bug. Being able to watch call counts rise as they interacted was enough to know whether the code they were looking at corresponded to the behavior they thought it did, and to get a sense of how often the code was executed, both of which breakpoints turned out to be poorly suited for. However, it seems that breakpoints are still needed when the

programmer wants to walk through the code, or when they want to inspect a lot of program state at a time.

S21's experience suggests that visualized data which merely supports a certain conclusion without proving it outright can be useful for making snap judgements without distracting programmers from their current task.

Four programmers wanted more visualizations with a similar presentation to the visualizations in Theseus, although not necessarily for them to be always on. The types of information desired were varied enough to suggest always-on display modes depending on the task. Users found problems such as the overhead of the instrumentation and information overload in the log, but call counts were used in a variety of ways, and the call counts and coloring turned out to be a good lightweight alternative to breakpoints for finding code correspondences and having "just enough" evidence to support a conclusion. The evaluations have revealed several avenues for improvement on a style of debugging interface whose time has come.

OPEN-SOURCE DEPLOYMENT

Theseus has been available as an extension for the Adobe Brackets editor since February 11, 2013. It can be installed from within Brackets, and the source code is available[5]. As of December 2013, Theseus has been installed approximately 2,500 times (users may choose not to send usage data) and 512 of those users have actively used Theseus on at least 2 separate days. 46 bug reports and feature requests have been filed.

CONCLUSION

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.

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. Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive record/replay for web application debugging. UIST '13, ACM (New York, NY, USA, 2013), 473–484.
2. Chilton, L., Kim, J., André, P., Cordeiro, F., Landay, J., Weld, D., Dow, S., Miller, R., and Zhang, H. Frenzy: Collaborative Data Organization for Creating Conference Sessions. SIGCHI '14 (2014).
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. *TOSEM 22*, 2 (2013), 14.
4. Gould, J. D. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies 7*, 2 (1975), 151–182.
5. Griffiths, D. Scheme Bricks, Sept. 2013. **http://www.pawfal.org/dave/index.cgi?Projects/Scheme%20Bricks**.
6. Ko, A., Myers, B., and Aung, H. Six learning barriers in end-user programming systems. In *VL/HCC* (2004), 199–206.
7. Ko, A. J., and Myers, B. A. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *SIGCHI '04*, vol. 6 (2004).
8. 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).
9. LaToza, T. D., and Myers, B. A. Visualizing Call Graphs. In *VL/HCC 2011* (Sept. 2011).
10. Lieberman, H., and Fry, C. Bridging the Gulf Between Code and Behavior in Programming. CHI '95 (1995).
11. Mankoff, J., Dey, A. K., Hsieh, G., Kientz, J., Lederer, S., and Ames, M. Heuristic evaluation of ambient displays. CHI '03, ACM (2003), 169–176.
12. McDirmid, S. Usable live programming. *SIGPLAN* (2013).
13. Meier, M. S., Miller, K. L., and Pazel, D. P. Experiences with Building Distributed Debuggers. In *Proc. SIGMETRICS 1996* (1996).
14. Microsoft. Debug Your App by Recording Code Execution with IntelliTrace. **http://msdn.microsoft.com/en-us/library/vstudio/dd264915.aspx**.
15. 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.
16. Saff, D., and Ernst, M. Reducing wasted development time via continuous testing. In *ISSRE '03* (2003), 281–292.
17. Schrock, E. Debugging AJAX in Production. *ACM Queue* (2009).
18. Sillito, J., Murphy, G. C., and De Volder, K. Questions programmers ask during software evolution tasks. SIGSOFT '06/FSE-14, ACM (2006), 23–34.
19. Swift, B., Sorensen, A., Gardner, H., and Hosking, J. Visual code annotations for cyberphysical programming. In *1st International Workshop on Live Programming (LIVE)* (2013).
20. Tanimoto, S. Towards a theory of progressive operators for live visual programming environments. In *IEEE Workshop on Visual Languages* (1990), 80–85.
21. Victor, B. Learnable Programming. **http://worrydream.com/#!/LearnableProgramming**, 2012.

[5]Theseus: **https://github.com/adobe-research/theseus**
Fondue: **https://github.com/adobe-research/fondue**