Theseus: Understanding Asynchronous Code

Tom Lieber
MIT CSAIL
Cambridge, MA 02139 USA
tl@csail.mit.edu

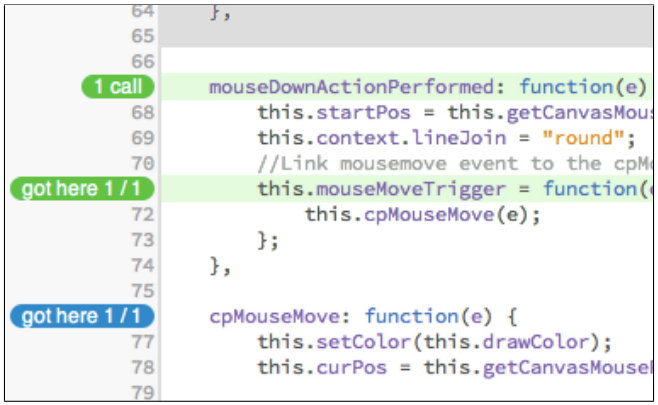


Figure 1: Theseus with two pseudo-breakpoints active

CHI 2013 Extended Abstracts, April 27–May 2, 2013, Paris, France.
ACM 978-1-4503-1952-2/13/04.

Abstract
The behavior of JavaScript is difficult to understand due to the language's asynchronous and dynamic nature. In particular, chains of event handlers pose difficulties because they cannot be stepped through with a debugger, and determining where a chain is broken requires instrumenting every link in the chain with a breakpoint or log statement. The aim of this work is to create a debugging interface that helps users understand complicated control flow in languages like JavaScript. Theseus uses *program traces* to provide real-time in-editor feedback so that programmers can answer questions quickly as they write new code and interact with their application. The call graph is augmented with *semantic edges* that allow users to make intuitive leaps through program traces, such as from the start of an AJAX request to its response.

Author Keywords
programming; debugging; code understanding

ACM Classification Keywords
D.2.5 [Testing and Debugging]: Debugging aids.

Introduction

In JavaScript on the web, and many other environments, event-driven and asynchronous code is common. Examples include handling events in a graphical user interface, requesting data from a network resource, and responding to HTTP requests. When these asynchronous operations depend on one another, the result is sometimes called 'callback hell'[1] and it can be difficult to understand due to its convoluted, non-linear control flow [6].

When seeking to understand code, programmers ask a multitude of questions [7]. Reachability questions, which concern control flow, are among the most difficult to answer without specialized tools [2]. An example reachability question is, "Are any network calls made directly or indirectly from this method?" Answering questions like these can be difficult without tools specifically designed for them. For instance, a step debugger would force the user to perform an in-order traversal of the call graph. Sequences of event handler registrations and activations (i.e. callback chains) are discontiguous in a program trace and are even more difficult to follow.

Theseus, the software presented in this paper, is a debugging interface for making sense of control flow and addressing the aforementioned difficulties. It uses two means to accomplish that task:

- It presents a streamlined interface for executing queries over program traces.
- It augments the call graph with *semantic edges* representing high-level connections such as callbacks.

Theseus uses code coloring and a breakpoint-like interface to answer developers' questions. Its information comes from program traces that are updated in real-time as the code is executed. It allows users to inspect the program's run-time state, similar to a step debugger. However, using program traces gives Theseus the ability to present aggregate information about state and control flow, and summarize past and future control flow from any given point. Users interact with the trace using *queries* consisting of an ordered set of *pseudo-breakpoints* they place in the code.

To address the problem of complex, event-driven control flow, Theseus augments the call graph with edges that correspond to high-level leaps through the program trace, such as from the start of an AJAX request to its response. Those *semantic edges* allow the user to walk along execution paths that are implied by the code structure, not just the path that is actually traversed by the computer.

Theseus is ongoing research. The interface is available as an extension[2] to the Brackets text editor[3]. It collects information by way of a proxy server that adds trace-collecting code to all JavaScript on a web page. A series of four prototypes has been created as part of an iterative design process. Several graduate students with asynchronous programming experience informally evaluated all four prototypes to guide the interface design. Three professional JavaScript developers at Adobe provided feedback on the fourth prototype. The fifth prototype, which is under development, is presented here.

[1] http://callbackhell.com/

[2] http://github.com/adobe-research/theseus/

[3] http://brackets.io/

1) Code that has yet to be executed is given a gray background.

2) Pills like 1 call appear next to functions to display the number of times they have been called. When the user clicks to add them to the query, they turn green.

3) Call counts become relative to the root of the query when one is active. The trace containing a call to `mouseDownActionPerformed` also contains a call to the `mouseMoveTrigger` callback, so it reads 'got here 1/1'.

4) The query automatically extends to every function in the program, making their call counts relative as well, reducing the need to click them.

5) Functions and page events that are part of the query show up here with the values of variables that were in scope, where they can be inspected.

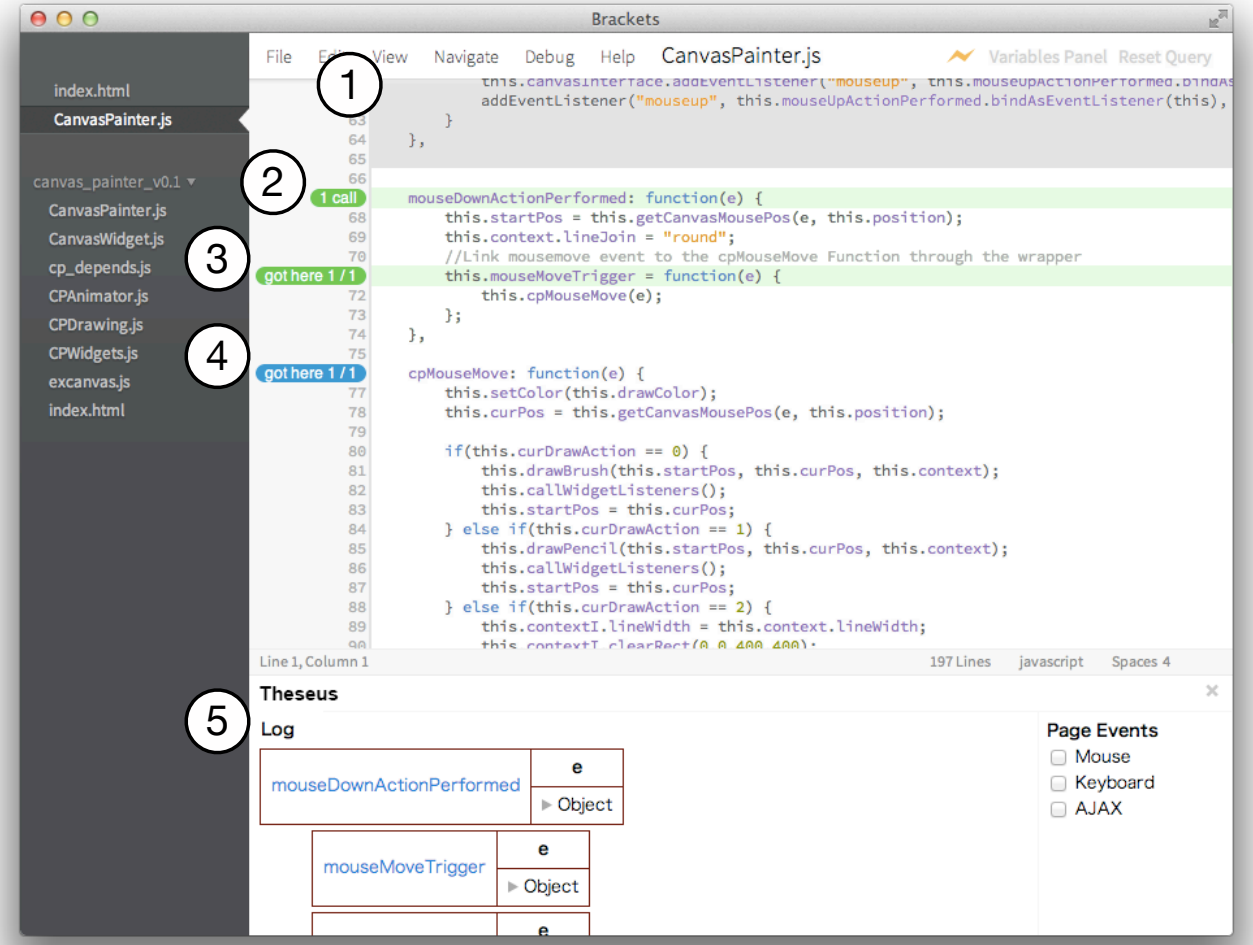


Figure 2: The Theseus user interface running as a Brackets extension

Example Usage Scenario

Theseus is designed to become part of the edit-compile-test loop by answering questions about run-time behavior as quickly as possible. In this section, we show how Theseus is a benefit to Max, whose task is to fetch search results with an AJAX request when a button is pressed.

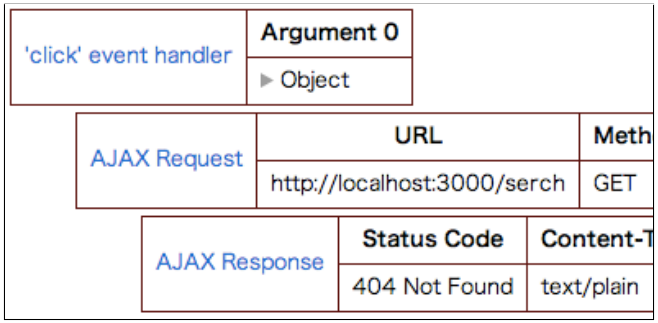
Max begins by connecting an event handler to a search button. Once he has entered the code to the best of his ability, he wants to verify that he made no mistakes. To do so, Max reloads the page, clicks the search button, and returns to the IDE. The coloring and call counts indicate that the code that registered the event handler, and the event handler itself, were both executed.

```
1 call      $(function () {
1 call          $("button").on("click", function () {
3               });
4       });
```

Next, Max needs to request the search results. He arrives at the code below by customizing a snippet he finds on the web. Max refreshes the page and clicks the search button again. From the coloring of the source code, he sees right away that the `success` event handler was never called.

```
1 call      $(function () {
1 call          $("button").on("click", function () {
3                   $.ajax("/serch", {
0 calls                 success: function () {
5                       }
6               });
7           });
8       });
```

To figure out why, Max checks the 'AJAX' box under 'Page Events' to add AJAX events to the Log. To focus on the AJAX events resulting from the search button, he clicks the 1 call pill next to the button's click handler. That adds the invocation of the click handler to the timeline and nests the related AJAX calls underneath.



Max sees that the AJAX request resulting from the button press returned a '404 Not Found' error, calling his attention to a typo in the URL (`serch` instead of `search`). He refreshes the page and verifies that the problem is fixed when all the call count pills turn blue.

To display the search results, Max needs to know how the data is passed to the AJAX callback. He clicks the 1 call pill next to the callback function. Theseus records the values of unnamed function arguments, which he inspects to discover that the first argument contains the desired data. He gives a name to that argument and writes the code to display it on the page.

```
1 call      $(function () {
1 call          $("button").on("click", function () {
3                   $.ajax("/search", {
1 call              success: function (data) {
5                       $("body").append(process(data.results));
6                   }
7               });
8           });
9       });
```

Discussion
Theseus provides several forms of feedback that would have been difficult or tedious for Max to gather without it. He is able to see whether lines of code had executed using color, instead of sprinkling the file with breakpoints, log statements, or calls to `alert()`. He is able to correlate page events, such as AJAX responses, with events in his code. He is able to inspect the values of variables without explicit instrumentation.

Theseus is able to associate AJAX events with the button click event by looking for the signs of the start of an asynchronous operation. In this case, Theseus detected the creation of an `XMLHttpRequest` object. Theseus will make a similar connection when it detects the creation of a closure, which is common for callbacks that are defined inline.

Interface Design

A session with Theseus starts when a user opens a web page from within the editor. When the user clicks one of the call count pills in the gutter of the code editor, that sets what we call a *pseudo-breakpoint*. A regular breakpoint halts execution when the breakpoint is reached so that program state can be inspected. A pseudo-breakpoint does not halt execution, but it retroactively adds all of the local state that could have been inspected to the Log panel at the bottom of the window.

In effect, setting a pseudo-breakpoint creates a query for all traces that pass through a particular point in the code. When the user sets another pseudo-breakpoint without clearing the first, that creates a query that matches traces that pass through both pseudo-breakpoints in sequence. This is analogous to halting at the first breakpoint, then stepping through the code to the second breakpoint. Theseus saves the user time by showing them much of the information they could have gathered from that process after two clicks.

Clicking the checkbox next to a page event is like setting a pseudo-breakpoint at all instances of that type of event.

Future Work

Theseus currently answers only a subset of the reachability questions that LaToza and Myers identified as difficult to ask, which I plan to correct in later prototypes [2]. In particular, Theseus queries require both ends of the query to be specified (except in the case of page events), making it unsuitable for finding traces that involve many functions.

The scenario presented in this paper contained several instances of the user needing to refresh the page in order to test changes to the code. I plan to extend Theseus to collect a more complete trace with which new code can be evaluated as if it had existed all along. That would provide users with the limited ability to use Theseus' reachability coloring and data inspection facilities without needing to reload the page. A more complete trace would also allow the user to evaluate arbitrary expressions in the log window, which would enable new styles of exploratory development similar to working at a read-eval-print loop (REPL).

As Theseus nears maturity, I am planning lab and field studies to evaluate its effectiveness. I hypothesize that Theseus will decrease the amount of time it takes developers to answer questions about run-time behavior of their code, leading to a decrease in the amount of time it takes to write code and fix bugs, and an increase in the success rate at fixing bugs. I will test this hypothesis

directly with lab studies using small programming tasks. I have also released the extension publicly to gather qualitative feedback from developers in the field.

Related Work

Theseus' interface was informed by research into the types of questions programmers ask while programming, 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.

Whyline [1] is a debugging interface users can ask questions like, "Why is this widget blue?" Whyline answers by generating a program slice of all the events that determined the line's color. Theseus, the work presented here, brings invisible state and behavior to the surface, providing more topics for users to ask about. For example, a grayed-out function is an opportunity to ask Whyline, "Why wasn't this function called?"

Reacher [3] answers reachability questions by presenting a compact graph representation of the interactions between several functions. The difference between Reacher's visualizations and those of Theseus stem largely from the difference in how data is obtained: statically versus dynamically. Theseus and Reacher provide complementary views of the same kind of information.

ZStep [4] is an omniscient step debugger that has commands for jumping around program traces similar to using *semantic edges*. A ZStep user can step forward and backward by line or expression, but also non-linearly, such as 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 [5] 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.

Acknowledgements

I thank my advisors Robert C. Miller and Joel Brandt for their guidance. 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 author's, and they do not necessarily reflect the views of the sponsors.

References

[1] 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).

[2] 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).

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

[4] 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).

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

[6] Myers, B. A. Separating Application Code From Toolkits: Eliminating the Spaghetti of Call-backs. In *Proc. UIST '91* (1991).

[7] Sillito, J., Murphy, G. C., and De Volder, K. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering 34*, 4 (July 2008).