Visualizing Call Graphs

Thomas D. LaToza Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
{tlatoza, bam}@cs.cmu.edu

***Abstract*—**Developers navigate and reason about call graphs throughout investigation and debugging activities. This is often difficult: developers can spend tens of minutes answering a single question, get lost and disoriented, and erroneously make assumptions, causing bugs. To address these problems, we designed a new form of interactive call graph visualization – REACHER. Instead of leaving developers to manually traverse the call graph, REACHER lets developers search along control flow. The interactive call graph visualization encodes a number of properties that help developers answer questions about causality, ordering, type membership, repetition, choice, and other relationships. And developers remain oriented while navigating. To evaluate REACHER'S benefits, we conducted a lab study in which 12 participants answered control flow questions. Compared to an existing IDE, participants with REACHER were over 5 times more successful in significantly less time. All enthusiastically preferred REACHER, with many positive comments.

Keywords-code exploration, call graphs, control flow, program visualization, program comprehension

I. INTRODUCTION

Control flow is one of the simplest and most expressive representations of a program. Control flow is often represented as a *control flow graph* which contains an edge from statement *a* to *b* when there exists an execution in which *b* executes immediately after *a*. In imperative programs, control flow expresses *causality* between a call site statement and a method[1]. Calling a method causes statements in it (and statements in methods it transitively calls) to execute. Determining when something happens requires finding the control flow by which it may be reached. And control flow expresses the *order* in which statements execute.

Developers work to understand a program's control flow throughout investigation and debugging activities as they mentally model, reason, and navigate [18]. For example, when investigating an unfamiliar codebase, developers first mentally construct a control flow representation of connections between its parts [20]. And their knowledge of a method's part of the call graph increases as they interact with its code [11]. Information foraging theory predicts that developers traverse control flow and search for "prey" – locations in code – by using "scent" – the similarity of the information which labels the control flow edges to their knowledge of their prey – to rank the potential of edges to traverse [19]. We have named

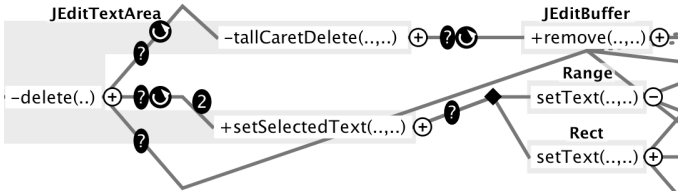

Figure 1. REACHER's call graph visualization supports reasoning about interprocedural control flow. For example, this visualization illustrates that JEditTextArea.delete(..) – on the far left – may call JEditTextArea.tallCaretDelete(.., ..) several times in a loop before it may call JEditTextArea.setSelectedText(.....) at two different call sites within a loop.

questions about what happens along a control flow path as *reachability questions* [18] because they ask whether certain code is *reachable* from other code under certain conditions.

Ensuring control flow is easily understandable has been an important goal of language design. Following Dijkstra's observation that `gotos` obfuscate control flow, making reasoning difficult [9], language designers introduced structured programming languages that simplify control flow within methods [4]. But in order to promote reuse and modularity, modern languages obfuscate interprocedural control flow between methods with features such as dynamic dispatch and indirection.

Interprocedural control flow is often visualized using a *call graph*. Modern Integrated Development Environments (IDEs) such as Eclipse and Visual Studio let developers see and navigate call graphs with commands ranging from *go to definition* (from a callsite) to providing a tree view for exploring call paths. Unfortunately, developers report that understanding control flow remains difficult [5][18]. As developers gain experience or learn a codebase, answering control flow questions becomes neither less frequent nor easier [18]. Developers often become disoriented and lost when exploring code [5]. Due to the difficulties of finding definitive answers, developers often guess, creating bugs when these guesses are incorrect [18]. Control flow paths which are long, widely branching, or contain inadequate or misleading scent can cause developers to spend tens of minutes answering a single question [18].

We designed REACHER to help developers more effectively answer reachability questions as they strive to understand and navigate control flow, find "prey," and stay oriented. REACHER automates searches along call graphs, freeing developers from manually traversing calls in search of statements. REACHER helps developers to understand control flow by depicting

[1] This paper uses the word "method" since our implementation is in an object-oriented language, but the techniques described here would apply to other imperative languages, where the words "function" or "procedure" might be used instead.

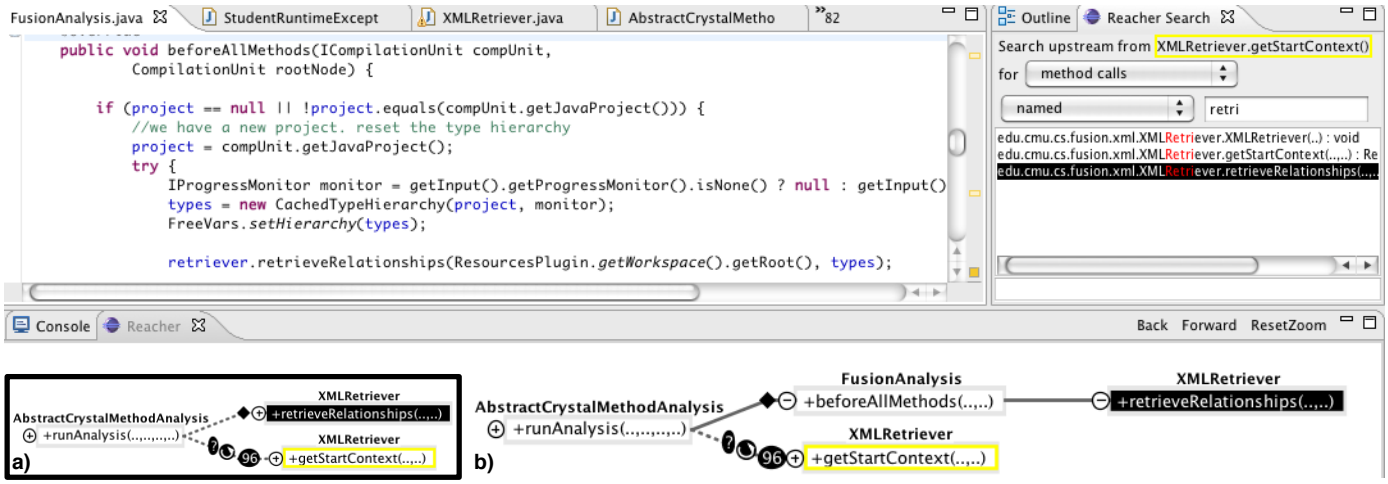

Figure 2. Can XMLRetriever.getStartContext() ever be called without XMLRetriever.retrieveRelationships() being called first? To answer this question in REACHER, a developer first opens XMLRetriever.getStartContext() in Eclipse. She right clicks the method declaration and invokes an upstream search. In REACHER'S search view (upper right), she types "retri". As she types, REACHER lists matching statements below. Clicking the third result adds it to REACHER'S call graph visualization (a). Looking at the visualization, she sees that all calls to getStartContext() are preceded by a call to retrieveRelationships(). But maybe there is a conditional somewhere on the path to retrieveRelationships()? Double clicking the path expands it (b), showing the method beforeAllMethods() which was previously hidden. Hovering over the call from beforeAllMethods() to retrieveRelationships() shows a popup describing the call (this edge is missing a ? due to a bug in REACHER). Clicking it opens the file in an Eclipse editor. Reading the code, she sees that the call is guarded by a conditional.

causality, ordering, type membership, repetition, recursion, and conditionality (see Figure 1). And REACHER helps developers remain oriented through its Eclipse plugin integration, letting developers open and read a method while still seeing its context in the call graph. To evaluate REACHER, developers used REACHER to answer reachability questions in several lab study tasks. Compared to Eclipse, developers with REACHER were over *5 times more successful in significantly less time*.

In this paper, we describe the iterative design and user testing of REACHER'S visualizations and interactions. Other papers present the extensive field studies that motivated this work [18] and the efficient static analysis used behind-the-scenes to implement REACHER [17]. In the remainder of the paper, we first illustrate the use of REACHER with an example. We then describe the design of REACHER, a lab study evaluating REACHER, related work, and conclude.

II. AN EXAMPLE

To see REACHER in action, consider a challenging debugging task we observed: a developer debugging a null pointer exception tried to understand how XMLRetriever.getStartContext() could ever be called without XMLRetriever.retrieveRelationships() being called first. Working in a codebase she had written herself, she spent 40 minutes answering this question, using the debugger to inspect values and statically browsing. The task was hard because 96 paths connected these methods, some as long as 13 calls. Manually navigating and making sense of these paths was challenging. REACHER makes this task easier by automating the search and visualizing the relevant portion of the call graph. We illustrate this with a scenario of how the developer might have instead worked using REACHER (see Figure 2). After opening XMLRetriever.getStartContext() in a Eclipse editor, she selects the method declaration and opens a context menu. She searches along paths *to* the selected method by selecting *search upstream*. Moving her cursor to the textbox in the REACHER Search view (upper right), she searches for connections to the other method – XMLRetriever.retrieveRelationships() – by typing "retri". As she types each character, REACHER lists matching statements below. Seeing retrieveRelationship() in the list, she clicks it, adding it to the call graph visualization below.

The call graph now contains 3 methods – XMLRetriever.getStartContext() (the origin method), XMLRetriever.retrieveRelationships() (the method she searched for), and AbstractCrystalAnalysis.runAnalysis() (Figure 2a). As this was an upstream search, REACHER looked for a common method calling both retrieveRelationships() and getStartContext() and found runAnalysis(), adding it to the call graph. Two edges emerge from runAnalysis() – one to retrieveRelationships() and a second to getStartContext(). The edge to retrieveRelationships() leaves runAnalysis() above the edge to getStartContext(), indicating it executes first. Inspecting the call graph, the developer learns that, in fact, all paths to retrieveRelationships() are preceded by a path to getStartContext(). But perhaps there is a conditional guarding the path to getStartContext() that might cause it not to be called? The dashed edge from runAnalysis() to retrieveRelationships() indicates that some of the path is hidden, so she double clicks to expand the path, revealing the previously hidden method beforeAllMethods() which connects runAnalysis() to retrieveRelationships() (Figure 2b). Hovering over the edge between beforeAllMethods() and retrieveRelationships(), she sees a popup describing the call. Clicking the edge navigates the Eclipse editor to the callsite. She then sees the cause of the bug – eight lines above the callsite is a conditional guarding the call. While correct for the rest of the body, it should not guard this call. Moving the call to getStartContext() outside the conditional block fixes the bug.

III. USER INTERFACE DESIGN

REACHER helps developers explore code by supporting searching, navigating, reasoning, and making sense of complex interprocedural control flow. REACHER's design resulted

from a user-centered, iterative design process. We first conducted extensive field studies of developers exploring code to understand the questions developers ask and the challenges they face [18]. Based on these studies, we designed a visualization which encodes the information we found to be most relevant to answering these questions. Recognizing that reading the underlying code is ultimately an important part of making sense of paths, our visualization design focused on quickly finding the relevant code, answering high-level questions about paths, and providing context, leaving details about the code itself to be inspected using the code editor. We built an initial mockup of our design and used a paper prototype study to further refine the details. We then implemented the visualization in the Prefuse visualization toolkit [12]. In the next sections, we describe REACHER's final user interface design, describing the rationale for several important decisions with references to previous alternatives we considered.

A. *Searching along control flow*

One of REACHER'S most important features is the ability to *search* along control flow. REACHER supports both *upstream* and *downstream* searches. An upstream search begins at a destination method and traces paths *upstream* by which it may be reached. *Downstream* searches begin at an origin method and trace paths *downstream* to its callees (and methods they transitively call). Downstream and upstream searches are asymmetric (see Figure 3). A downstream search captures what an origin *does* – all of the causality relationships linking the origin to other methods, directly or indirectly. Searching upstream finds what happens *before* a destination, including both direct and indirect callers and methods called before by direct or indirect callers. These types of searches correspond to the two most frequent types we observed in our studies [18].

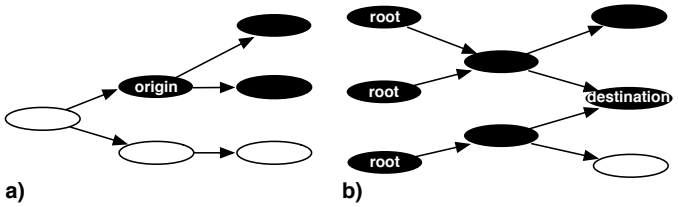

Figure 3. a) A *downstream* search finds methods (shaded ovals) on paths from an origin method, but does not find methods on paths returning from the origin (unshaded). b) *Upstream* searches find method paths terminating at a destination, including other methods that are called.

REACHER searches along a static, conservative approximation of paths that may execute (see [17] for full details of the analysis algorithm). An alternative approach would be a dynamic approach in which the user runs the program and enters input to demonstrate the situation of interest, and the tool records an execution trace (c.f., [16]). A key advantage of a dynamic trace is that it is fully precise, containing no false positives – only statements that actually executed are included. But a static approach such as REACHER'S enjoys several advantages. A static approach permits reasoning about everything that could possibly happen, which may not be evident from a single trace or even from many traces. Generating a dynamic trace is annoying for situations that are difficult to reproduce, time-consuming for long running operations, difficult when special hardware or configuration is required, and impossible when the input necessary to cause the desired path to execute is unknown. For example, when debugging field-reported failures with only stack dumps to indicate the problem, developers may not know how to generate such a stack. In our field observations of professional developers [18], dynamic tracing would not have worked in two-thirds of the longest tasks involving reachability questions. Thus, developers with current tools explore code with a combination of dynamic and static approaches.

One drawback of a static approach is false positives—*infeasible* paths that never execute due to correlations between conditionals. In codebases with extensive use of message passing or dynamic dispatch, this can be particularly problematic, connecting portions of the codebase that are not in fact connected. In order to mitigate this problem, REACHER uses fast feasible path analysis (FFPA)[17] to eliminate some of the most common forms of infeasible paths. FFPA constructs summaries describing possible paths through a method. When the user initiates a search, an interprocedural dataflow analysis uses these summaries to propagate constants, partially-path sensitively, and determine which branches through conditionals are feasible. In most common cases, FFPA is able to generate paths in one to two seconds of analysis time.

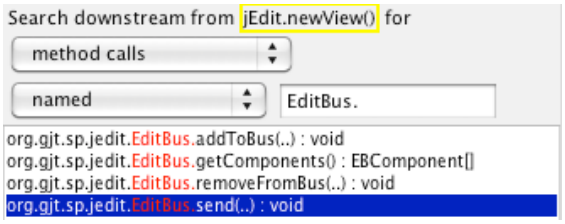

Figure 4. REACHER lists matching statements as users enter each character of a search. Double clicking a result *pins* it, assigning the corresponding search a unique color and persistently adding the selected item to the call graph visualization.

REACHER provides a search view for describing searches along control flow (Figure 4). REACHER indicates if the search is downstream or upstream and the origin or destination method. Users can select both what kind of item to search for (method, library, or constructor calls; field read, writes, or accesses; or any of these) and which parts of the name to match (package name, type name, or type and method name). Our field observations of reachability questions contained several examples of searches scoped to a specific type of method or statement [18]. Search text may match any portion of the identifier, not just the first portion. The matching portion of the result is highlighted in red. These features make it easy to find a target by knowing just a fragment of a name or relevant concept while also minimizing typing. Selecting a result adds it to the call graph. Selections are ephemeral, supporting quick scrubbing to visualize each result in turn. Double clicking a result *pins* the item, persisting it in the visualization.

REACHER lists search results – methods and fields – with their fully qualified name and type. We experimented with instead showing a portion of all matching statements. For example, searching for foo() might display *multiple* callsites such as a.foo() and b.foo(). Searching for fields included every access and assignment statement. This provided more context

and made it possible to select individual callsites and access statements. But this context made the text for each result much longer, making the result list wider and occupying more space. Additionally, result lists were far longer – methods called frequently could be included tens or hundreds of times rather than once. And forcing users to choose a specific callsite or field access statement was more distracting than helpful. So REACHER lists each method or field only once. After selecting a result, users see the context in the call graph visualization.

B. *Methods and expressions*

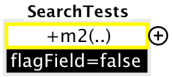

Figure 5. REACHER's depiction of the method SearchTests.m2(boolean) and the field write flagField = false.

REACHER visualizes call graphs as graphs of method nodes and call edges. Following the UML's conventions [23], public, protected, and private methods are prefixed by +, #, and -, respectively. The identifiers of static methods are italicized. To help distinguish overloaded methods, each parameter is indicated with a "..", and parameters are separated by commas. Including the parameter names and types would be unambiguous, but, even for common cases, names become several times longer, with a corresponding reduction in the number of methods shown in a fixed space. When a selected search result is a field access or a library call, REACHER displays the field access expression or callsite statement below the method in which it is located (see Figure 5). The method the user searched from is highlighted with a yellow box, corresponding to the yellow box in the search window (Figure 2, upper right).

Previous research shows that developers often get disoriented when trying to explore the control flow to and from a method [15][5]. REACHER helps with this problem by working as a navigation aid – clicking a method in the callgraph opens the code in an Eclipse editor.

C. *Causality*

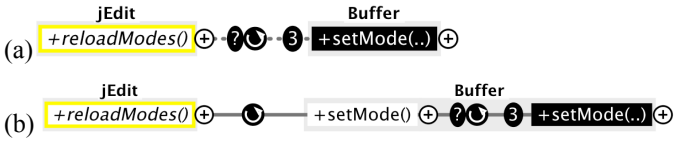

Figure 6. (a) Indirect calls (dashed lines) expand into (b) one or more paths of direct calls (solid lines).

Causality is a central part of reachability questions – what does this do and when does it happen? REACHER'S call graph is designed to help developers reason about causality. When a method node is created in the call graph, REACHER finds all of the control flow paths connecting it to existing nodes in the call graph, showing all of the ways it might be triggered. Knowing there *is* a causal relationship is often sufficient, so REACHER displays these control flow paths as a single indirect call edge (Figure 6a). These paths are often long, complex, and uninteresting; hiding them significantly reduces irrelevant clutter. When the path is interesting, developers can double click it, expanding it to show the previously hidden methods in the path (Figure 6b). Clicking a call edge navigates the editor to the corresponding call site.

While searching helps to locate distant methods, developers sometimes explore a method's immediate callers and callees. For downstream searches, REACHER depicts a circled plus icon ⊕ when a method has hidden *callees*. Clicking the icon expands all of the callees, changing the icon into a circled minus icon. Clicking the minus icon hides the callees. Similarly, for upstream searches, REACHER provides a plus icon to the left of the method indicating that there are hidden *callers*.

D. *Ordering*

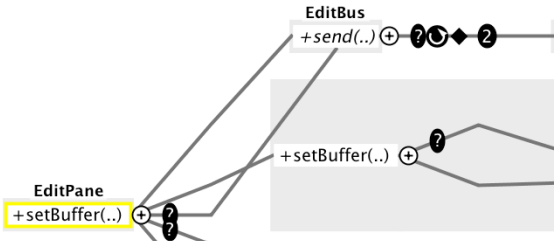

Figure 7. Outgoing calls execute from top to bottom.

In most of the exploration tasks we observed, developers used information about the *order* of calls. Therefore, unlike existing call graph visualizations, REACHER visually encodes the call order, sorting outgoing edges in execution order from top to bottom (see Figure 7). This unambiguously orders paths through the call graph. To distinguish incoming from outgoing edges, edges exit a method from the right and enter from the left. When there are multiple incoming edges, all but the first enter from the bottom to help disambiguate multiple incoming edges.

Upstream searches cause additional complexity when a user adds a method *m* that executes before any visible methods. As REACHER'S edges denote indirect or direct calls and no currently visible method calls *m*, no edges connect it, and its order is not visible. To solve this problem, REACHER computes the least upper bound method between *m* and currently visible methods. A least upper bound must exist for *m* to be upstream. The least upper bound is then added to the call graph. For example, after adding getStartContext() and retrieveRelationships(), REACHER adds the least upper bound runAnalysis() (see Figure 2), showing that getStartContext() executes before retrieveRelationships().

REACHER use a single node for methods along all paths by which they are reached, connecting each path after the first with backward edges. For example, in Figure 8, tallCaretDelete() and Range.setText() both call remove(), with a backward edge to remove() denoting setText()'s call. Backward edges increase visual complexity, introducing non-tree edges that overlap and cross. We considered instead creating a tree structure by replicating repeatedly called methods, except for recusive calls. However, replicating not only replicates the method itself but also its entire subtree of direct and indirect callees. Replicating subtrees greatly increases the call graph's dimensions. For example, expanding with replication the path in Figure 2 between runAnalysis() and getStartContext() increases the number of rows from 8 to 97. Furthermore, replication makes understanding subtrees more

challenging by forcing developers to manually compare nodes between similar subtrees to identify differences.

However, using a single node for each method increases visual complexity, creating overlapping and crossing edges that can be challenging to untangle. To help solve this problem, REACHER lets developers mouse over an element to see its connections (see Figure 8). Entering a node highlights incoming and outgoing edges; entering an edge highlights incoming and outgoing nodes. One study participant commented, "It kinda reminds me of a magician, that if they want to see if there are any wires around they move their hand."

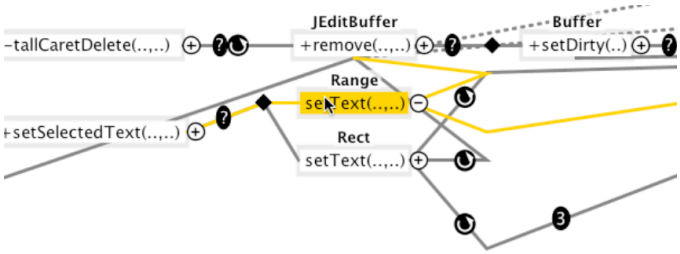

Figure 8. Mousing over a method highlights incoming and outgoing calls.

E. *Type membership*

Types (e.g., classes) express a developer's intention that the methods and fields they contain are related. REACHER visually encodes type membership with shadows grouping adjacent methods with a common type (see Figures 1, 6, and 9).

F. *Layout*

REACHER uses an automatic layout to assign each method a position. REACHER'S layout technique begins at *root* methods – methods with no visible callers. Call graphs produced by upstream searches may have multiple roots. From each root, REACHER computes a spanning tree. For methods with multiple incoming edges, the spanning tree includes the edge which executes first. REACHER then walks the spanning trees in-order to compute positions for each method, assigning positions from top to bottom and left to right. For methods with a single callee, both are assigned to the same row, with the caller to the left of its callee. For methods with multiple callees, each callee is given its own row from top to bottom. This process hierarchically computes a row and column assignment for each method. Row height and column width are then assigned using the maximum vertical and horizontal dimensions, respectively, of their cells. Finally, REACHER stacks each spanning tree vertically, with backward edges linking trees.

G. *Repetition and choice*

Realizing that a call is guarded by a conditional or may execute repeatedly can be important for answering reachability questions. REACHER alerts developers to the presence of these constructs by visualizing repetition and conditionals with call edge icons. Question marks indicate a conditional guarding a call's execution; loop icons indicate callsites in a loop. When a call could be to one of several overriding methods because of dynamic dispatch, edges to these callees begin with a single shared line and branch into separate lines at a diamond icon. REACHER condenses repeated edges to the same method into a single edge, indicating the edge count with a number icon. But when an edge to a different method is interleaved between the repetition, the repeated edges are shown separately before and after the interleaved edge, showing ordering. For example, in Figure 7, the repeated calls to send() are shown before and after the interleaved call to setBuffer(). Hovering over an icon displays a descriptive popup (see Figure 9).

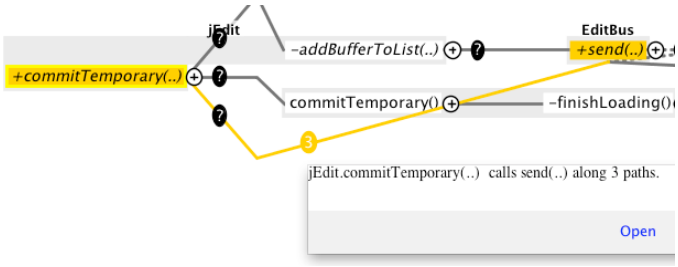

Figure 9. Hovering over an icon or edge displays a descriptive popup.

H. *Supporting rapid exploration*

REACHER provides a variety of additional interactive features for rapidly expanding details and then hiding them again if the user decides they are not relevant. "Back" and "forward" commands traverse a web-browser style navigation stack of visualization states. Pan and zoom commands lets users focus on specific areas or get an overview. To help users track the location of methods as new methods are added and layout positions change, REACHER smoothly animates transitions. Showing the callers or callees of a method anchors the method's position, moving other nodes relative to it.

IV. EVALUATION

REACHER'S design is premised on the assumptions that searching along control flow is faster than traversing paths using conventional navigation techniques, and that visualizing paths can help developers more effectively understand and navigate code. We conducted a lab study to test these assumptions, and evaluate the potential productivity benefits of REACHER and the usability of REACHER's features.

A. *Method*

12 participants were recruited from students and staff at Carnegie Mellon University. All participants reported being comfortable programming in Java (median = 4.5 years experience), had professional software development experience (median = 1.1 years), and knew an average of 4 programming languages. None had previously used REACHER.

Participants performed 6 tasks and were given 15 minutes to complete each task. Each task posed a reachability question and involved finding and understanding control flow between events. Table 1 lists each of the tasks' questions. To test if participants were able to understand the visualization notation, each task was designed to require understanding a particular aspect of the notation. Tasks 1 and 2 dealt with ordering, tasks 3 and 4 dealt with conditions, and tasks 5 and 6 dealt with repetition. All participants performed all 6 tasks and did half of the tasks with Eclipse alone and half with Eclipse and REACHER. Participants were randomly assigned to conditions. The order of the tasks, whether they received the 3 Eclipse only tasks or the 3 REACHER tasks first, and which tasks were used in each condition were all counterbalanced.

Task 1. When a new view is created in jEdit.newView(View), what messages, in what order, may be sent on the EditBus (using EditBus.send())?
Task 2. When text is deleted (JEditTextArea.delete()), what is the first message that may be sent on the EditBus (using EditBus.send())?
Task 3. Does setting the buffer in EditPane.setBuffer() cause the caret status on the status bar to be updated at least once (StatusBar.updateCaretStatus())?
Task 4. Other than the check that the firstLine has changed from the oldFirstLine in setFirstLine(), are there other conditionals that might cause JEditTextArea.setFirstLine() not to update the scroll bar (JEditTextArea.updateScrollBar())?
Task 5. How many messages may jEdit.commitTemporary() send to the EditBus? (i.e., how many times might it invoke EditBus.send()?)
Task 6. How many messages may jEdit.reloadModes() send to the EditBus? (i.e., how many times might it invoke EditBus.send()?)

Table 1. Participants were asked to answer a series of six reachability questions.

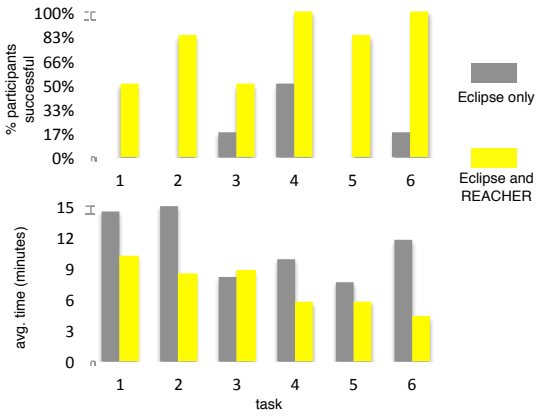

Figure 10. Success and average task time. Task time includes participants that failed. Participants who ran out of time received 15 minutes.

All tasks were performed in the jEdit codebase, a 55 KLOC open source text editor used in several previous studies of code exploration [22][6][18]. Several of the tasks dealt with jEdit's EditBus which provides a publish / subscribe mechanism for sending and receiving messages. Participants were asked questions such as what events were sent on the bus or to trace messages through the bus.

To ensure all participants were familiar with Eclipse's many code navigation features, all participants were first given a tutorial on Eclipse (adapted from [22]). Before performing tasks with REACHER, participants completed a second tutorial that explained the notation and interactions and in which they used REACHER to answer a sample reachability question. Participants were given task instructions on paper and allowed to take notes. Participants used Eclipse 3.6.1 and were allowed to use any feature they wished. Participants worked on a 2.8 Ghz computer with 8 GB of memory, a large 30" monitor, and an additional laptop screen. To understand why developers used the approaches they did, participants were asked to think aloud, and we recorded audio and the screen with Camtasia.

B. *Results*

Participants completed tasks 5.6 times more successfully with REACHER (78%) than with Eclipse alone (14%). Averaged across all tasks, participants' mean task time was 11.1 minutes with Eclipse alone and 7.2 minutes with REACHER. This is a conservative estimate of the time difference, because we used a time of 15 minutes (the maximum) for tasks on which participants ran out of time, whereas they would likely have taken much longer. Figure 10 shows success and task time per task. Participants were significantly faster with REACHER in tasks 1, 2, 4, and 6 ($p < .05$), but not tasks 3 ($p = .6$) or 5 ($p = .25$). Participants succeeded too infrequently with only Eclipse to compare times between just those who succeeded.

Participants with only Eclipse used a number of static exploration strategies. When reading a method, participants relied heavily on the "scent" of method names at call sites to decide which methods to open and read. For example, to find paths to EditBus messages, participants reasoned about which methods might be likely to do something requiring an EditBus message to be sent. Some participants tried to methodically traverse many paths, while others guessed which would be most likely to lead to the target. Many participants explicitly debated whether it was better to guess or methodically explore. Most participants also navigated to the target statement to get a sense for what it did and when it might be likely to happen.

Most participants with only Eclipse used the call hierarchy to traverse paths of calls. But, due to the huge fanout of methods, most realized the hopelessness of finding their target method in this view. Several participants did bidirectional search, navigating call hierarchy paths both forwards and backwards and trying to pick methods to traverse based on similarity to calls from the other direction. A significant barrier to static traversal were event listeners, implemented using the Observer Pattern. To determine which methods were actually called, participants would have to determine which classes implemented the interface and then begin new traversals from these methods. This forced them to perform new call hierarchy searches, losing their place. Participants sometimes said that trying to discover a listener was disheartening, as it signified there was much more to understand.

One participant tried to use dynamic, rather than static, investigation, and faced different challenges. To use the debugger to investigate a method, he first had to find a user command which invoked it, and he statically traversed upstream using the call hierarchy. After finding a command, he ran the program and invoked it, but found that conditionals prevented the path he wanted to see from executing. Returning to static investigation, he tried to find when they were true. But even after figuring out how to invoke the functionality, he faced a further challenge. To find paths from an origin to the target, he breakpointed the target, repeatedly hit the breakpoint, and investigated the paths. But as the target was widely called by methods other than the desired origin, many of the times that the breakpoint was hit were not paths from the origin. While he tried to only investigate those paths from the origin, he occasionally forgot to check and investigated the wrong paths.

All participants began using REACHER by opening the origin method described in the task, invoking a downstream search, and expanding the resulting paths. While participants often had a correct answer early in the task, they then spent most of their time better understanding the code to be sure of their answer, using REACHER to navigate to callsites along the path and discover what the calls were doing. Several attempted to more precisely determine in which situations different paths may execute by inspecting conditionals and trying to

understand when they might be true by tracing the data that flowed into them.

As participants read methods in the editor, REACHER'S call graph provided context and helped them to stay oriented:

I like it a lot. It seems like an easy way to navigate the code. And the view maps to more of how I think of the call hierarchy.

It seems pretty cool if you can navigate your way around a complex graph.

Without REACHER, participants were often disoriented:

Where am I? I'm so lost.

I think I lost where I am in this silly tree.

There was a call to it… somewhere, but I don't remember the path.

All participants reported that tasks with REACHER were easier; most had strongly positive impressions:

REACHER was my hero. … It's a lot more fun to use and look at.

It's very cool actually. You don't have to ... go through many, many files.

Oh, this is really great, how do you find this stuff [methods along paths]?

It seems really useful.

You don't have to think as much.

Many felt that tasks without REACHER were very difficult:

Ah, this is going to get miserable isn't it.

This is pretty ugly.

Failing tasks while using REACHER was infrequent (22%) but not absent. 6 of the 8 failures were in tasks 1 and 3. Some of these failures were caused by failing to find all of the paths due to overlapping edges or paths that zigzagged through the graph. Others were caused by participants focusing on part of a path and missing an icon on the rest of the path. For example, one participant failed task 3 because they missed a ? icon at the end of a long path. Even for participants that succeeded, following paths was hard. One participant suggested highlighting the path from the current node to a root.

Our study revealed a number of other usability problems. Edges that passed through methods or overlapped were initially confusing until users discovered the highlighting feature. Some participants found it difficult to visually locate targets in the call graph. While these methods are already rendered using a distinctive black fill and white text, participants suggested making them even more easily recognizable. Participants failed to notice that incoming and outgoing edges intersect nodes at different positions but instead relied on popups to disambiguate the direction of backward edges. One participant suggested indicating edge direction with arrows. A few participants wished to disentagle cluttered visualizations by dragging methods and manually overriding their layout positions.

C. Limitations

Our study had several limitations. By phrasing the task instructions as reachability questions, we did not include the surrounding debugging or investigation task context which normally motivates users to ask these questions. While participants felt that searching along control flow was representative of their actual work, several felt that questions about path attributes (e.g., how many times…) were contrived. We included these questions to make sure that our visualization was clear and usable. Unlike most developers in the field, our participants had no experience in the codebase. Developers with more knowledge might more successfully predict where they should navigate. However, while these limitations may bias the magnitude of differences in our results, it should be remembered that studies have found that answering reachability questions is frequent and time-consuming in the field [18].

V. RELATED WORK

A number of previous systems have been designed to visualize call graphs. For example, Rigi provides an extensible framework for visualizing graph structures during reverse engineering tasks [21]. It provides interactive tools, such as fisheye views, for exploring graphs. However, Rigi provides no support for searching, does not hide paths inside indirect calls, and does not depict ordering, choice, repetition, loops, or statements.

Many previous systems have been designed to help developers more effectively explore code. Some systems build a graph of elements connected through relationships and let developers traverse paths through these relationships. Relationships which these systems have explored include static slices (e.g., CodeSurfer [1]) and dynamic slices recorded from execution traces (e.g., WhyLine [16]). JQuery [14] traverses structural relationships amongst types and methods (e.g., method membership, subtyping, containment, references, constructors), providing a unified tree view including both methods and types. However, many of these tools have never been evaluated in a lab study. One of the few such studies evaluated JQuery and two other code exploration tools with code exploration tasks in jEdit (as in our study) and found no significant benefits from any of the tools [6].

While most of these systems do not support searching along paths, a few do. In Dora [13], developers select an origin method and enter a search string, and then may inspect a graph depicting call graph paths to methods textually similar to the search string. However, using Dora to answer reachability questions would be challenging. It does not support searching for field reads, field writes, or library calls or searching for methods in specific types or packages, making it impossible to directly express most of the reachability questions we observed in our field research [18]. And Dora provides only a rudimentary call graph view. Dora's focus is instead on exploring the use of information-retrieval techniques in searches and is therefore complimentary to REACHER.

Diver provides limited support for searching along dynamic traces [2]. Diver lets developers search along an execution trace for method calls and visualizes traces as UML sequence diagrams. But, in Diver, searches are used only to locate methods, not to scope the visualization to search results. In situations where dynamic analysis is possible and helpful, dynamic traces could complement REACHER's static traces by

providing certainty of a path's feasibility and supporting inspection of concrete values.

Several systems have explored approaches for reducing disorientation during code exploration. Relo [25], Code Canvas [8] and Code Bubbles [3] help developers to stay oriented by providing a *map* of code. Replacing a conventional editor in which developers edit in a full size window, methods are instead shown in many small *bubbles*, providing context during reading and making it easier to rapidly switch between related methods. Like these tools, REACHER's visualization is intended to help minimize disorientation by letting developers select task relevant methods and visualize relationships among these methods. One important difference is that REACHER shows only method names and task relevant statements rather than the entire method's implementation. This makes REACHER'S visualization substantially more compact, allowing developers to simultaneously view many more methods. REACHER'S design may more effectively support situations in which developers investigate relationships between small snippets scattered across many methods. Moreover, both visualization styles could be incorporated in the same system by letting developers zoom in to see method's implementation and zoom out to see additional context.

VI. CONCLUSIONS

Our results demonstrate that REACHER helps developers explore code more easily and effectively, transforming a tedious, frustrating, disorienting, guess-work-filled task into one which most participants finished successfully. Our tasks effectively replicated the challenges exploring code that studies have repeatedly found developers face – finding methods, staying oriented, and understanding paths – and demonstrated that a combination of search, task specific visualization, and IDE integration makes code exploration significantly easier. REACHER'S most significant benefit is search, which helped developers more quickly locate far-away methods and statements connected by long and confusing paths. But REACHER also helps support the subsequent work of understanding and reasoning about the path. Participants traced call graph paths to identify properties of paths. Participants ultimately wanted to see the code behind these paths, and used REACHER to quickly jump between methods on the paths.

ACKNOWLEDGMENTS

We thank the participants of our study and Andy Ko, Polo Chau, Ciera Jaspan, Jonathan Aldrich, Andrew Begel, Rob DeLine, and Niki Kittur for helpful suggestions and discussions. This research was funded in part by the National Science Foundation, under grant CCF-0811610.

REFERENCES

[1] Anderson, P. and Teitelbaum, T. (2001). Software inspection using CodeSurfer. In *Proc. Workshop on Inspection in Software Engineering at CAV*.

[2] Bennet, C. D. Myers, Storey, M.-A., and German, D. (2007). Working with 'monster' traces: Building a scalable, usable, sequence viewer. In *Proc. of the Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, 1-5.

[3] Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, J. J. (2010). Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proc. Int'l Conf. Software Eng (ICSE),* 455-464.

[4] Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. (1972). *Structured Programming*, Academic Press, London.

[5] de Alwis, B., and Murphy, G. C. (2006). Using visual momentum to explain disorientation in the Eclipse IDE. In *Proc. Visual Languages and Human-Centric Computing (VL/HCC)*, 51-54.

[6] de Alwis, B., Murphy, G.C., and Robillard, M.P. (2007). A comparitive study of three program exploration tools. In *Proc. Int'l Conf. on Program Comprehension,* 103-112.

[7] DeLine, R., Khella, A., Czerwinski, M, and Robertson, G. (2005). Towards understanding programs through wear-based filtering. In *Proc. of the ACM Symposium on Software Visualization*, 183-192.

[8] DeLine, R., Venolia, G., and Rowan, K. (2010). Software development with code maps. In *Commun. ACM,* 53, 8 (Aug. 2010), 48-54.

[9] Dijkstra, E.W. (1968). Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (March 1968), 147-148.

[10] Fritz, T., and Murphy, G.C. (2010). Using information fragments to answer the questions developers ask. In *Proc. Int'l Conf. Software Eng (ICSE)*, 175-184.

[11] Fritz, T., Murphy, G.C., and Hill, E. (2007). Does a programmer's activity indicate knowledge of code?. *In Proc. of ESEC-FSE,* 341-350.

[12] Heer, J., Card, S. K., and Landay, J. A. (2005). Prefuse: a toolkit for interactive information visualization. In *Proc. Conference on Human Factors in Computing Systems (CHI),* 421-430.

[13] Hill, E., Pollock, L., Vijay-Shanker, A K. (2007). Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. Automated Software Engineering (ASE)*, 14-23.

[14] Janzen, D. and De Volder, K. (2003). Navigating and querying code without getting lost. In *Proc. Aspect-Oriented Software Development (AOSD),* 178-187.

[15] Ko, A. J., Aung, H., and Myers, B. A. (2005). Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proc. Int'l Conf. Software Eng (ICSE),* 126-135.

[16] Ko., A.J., and Myers, B.A. (2009). Finding causes of program output with the Java WhyLine. In *Proc. Conference on Human Factors in Computing Systems (CHI)*, 187-196.

[17] LaToza, T.D. (2011). Answering reachability questions. Dissertation, Institute for Software Research, Carnegie Mellon University.

[18] LaToza, T.D. and Myers, B.A. (2010). Developers ask reachability questions. In *Proc. Int'l Conf. Software Eng (ICSE).*In *Proc. ICSE,* 185-194.

[19] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K. (2011). How People Debug, Revisited: An Information Foraging Theory Perspective. In *Transactions on Software Engineering (TSE), to appear.*

[20] Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. In *Cognitive Psychology*, Vol. 19, 295-341.

[21] Storey, M.-A.D. and Muller, H.A. (1995). Manipulating and Documenting Software Structures Using Shrimp Views. In *Proc. Int'l Conf. Software Maintenance (ICSM)*.

[22] Robillard, M. P., Coelho, W., and Murphy, G.C. (2004). How effective developers investigate source code: an exploratory study. In *Transactions on Software Engineering (TSE),* 30(12), 889-903.

[23] Rumbaugh, J., Jacobson, I., and Booch., G. (1998). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[24] Sillito, J., Murphy, G.C., and De Volder, K. (2008). Asking and answering questions during a programming change task. In *Transactions on Software Engineering (TSE),* 34, 4 (July 2008), 434-451.

[25] Sinha, V., Karger, D., and Miller, R. (2006). Relo: helping users manage context during interactive exploratory visualization of large codebases. In *Proc. Visual Languages and Human-Centric Computing (VL/HCC)*, 4-8.