Visualizing Call Graphs

Thomas D. LaToza & Brad A. Myers, 
VL/HCC 2011

Summary by Prof. Thomas LaToza 
SWE 795, Spring 2017 
Software Engineering Environments
Motivation: Understanding control flow is hard

• Answering reachability questions frequent challenge in debugging & investigating implications of code
  - Error prone
  - Frequent
  - Hard
  - Time consuming
  - Caused 50% of bugs
  - >9 times a day
  - 82% agree
  - Not easier or less frequent with knowledge or expertise

• Underlying cause: making foraging decisions across calls

...
Searching along call graphs

Many methods, some of them are task relevant
Finding them is hard...
Information foraging models whole debugging / investigation task as traversing relationships to find search targets (prey) [Lawrance+2011]

But developers search for statements by attribute (e.g., field writes) and partial name.
## Design requirements for code exploration

<table>
<thead>
<tr>
<th>Finding</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>search for statements by <strong>attribute</strong> (e.g., field writes) and <strong>partial</strong> name.</td>
<td>Configurable search dialog, incrementally match statements</td>
</tr>
<tr>
<td><strong>rapidly</strong> investigate, never returning to most methods.</td>
<td>Expandable details on demand, browser style history navigation</td>
</tr>
<tr>
<td>explore <strong>huge</strong> call graphs, but task relevant portion small.</td>
<td>Only show the (task relevant) methods developers select.</td>
</tr>
<tr>
<td>reason about causality, class membership, ordering, choice, repetition.</td>
<td>Overview this information in visualization of callgraph</td>
</tr>
<tr>
<td>get <strong>lost</strong> and disoriented reading through code in disparate places.</td>
<td>Link callgraph to editor to navigate code.</td>
</tr>
</tbody>
</table>
Existing tools don’t solve the problem

**Graph visualizations**

- SHriMP [Storey+95]
  - not task specific
  - no search
  - no ordering, class membership....

**UML Sequence Diagrams**

- Diver [Bennet+07]
  - not task specific
  - not compact

**Maps of code**

- Code bubbles [Bragdon+10]
  - can’t search over paths
  - don’t compactly encode ordering, repetition, conditionals, ...
Reacher

Designed a tool for understanding, exploring, and reasoning about call graphs

 Implemented as an Eclipse plugin for Java
 Generates static call graphs with fast feasible path analysis
 Visualization built on Prefuse visualization toolkit \[\text{[Heer}+05]\]

Helps to

find statements
understand call graphs
stay oriented

by

entering searches
visualizing results, encoding properties
navigating IDE
Results

Developers with Reacher **5.6** times more successful than working with Eclipse only.

Participants with Reacher took an average of **7.2** minutes vs. **11.1** minutes with Eclipse only (difference limited by ceiling effect).

* significant differences (p < .05)
Control group traversed paths

• **Traversed** paths through code looking for targets
  Relied heavily on **scent** - perceived relevance of method on path
  E.g., to find EditBus messages, looked for important actions
  Traversing through event listeners forced new search, often lost place

• Sometimes did **bidirectional** search
  Started at origin and hypothesized destination
  Tried to find connecting paths

• **Dynamic** investigation was difficult
  Ran the program, but conditionals guarded path of interest
  Did **static** investigation to figure out how to dynamically execute
  But then was hard to determine which of many breakpoints hit it
Questions for Discussion

• Would you use this tool?

• In what contexts might Reacher be difficult to apply?
  • How might Reacher be extended?

• What are the pros and cons of static analysis vs. dynamic for debugging?

• What challenges might there be in commercializing Reacher?