Searching Across Paths

Thomas D. LaToza
Institute for Software Research
Carnegie Mellon University
tlatoza@cs.cmu.edu

Brad A. Myers
Human Computer Interaction Institute
Carnegie Mellon University
bam@cs.cmu.edu

ABSTRACT

Observations of developers indicate that developers try to answer a variety of questions by searching across control flow paths through a program for statements matching search criteria. We believe that tools that better support this activity can help developers answer these questions more easily, quickly, and accurately.

Categories and Subject Descriptors

D.2.6 [**Programming Languages**]: Programming Environments; D2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Code navigation, program comprehension, developer questions, software maintenance

1. INTRODUCTION

While working on this paper, the first author was interrupted by a friend looking for advice. She was considering several possible alternative changes and was wondering which was the best design. In the midst of this discussion, a question emerged: would an object with a null constituent part break some existing code? If not, then a simple solution was possible. Breaking from the discussion, she went off to explore the code. From experience, she knew the method usually used to create such objects, which it did by calling into a factory used to create one of many possible subtypes. Much of the work of these methods was to set up the constituent parts. She hypothesized that maybe objects already could be created with a null constituent part, but since such objects do not need this initialization, were created by a different path. She searched for references to constructors of the subtypes. Unfortunately, there were many subtypes that were mostly called by the paths from the factory that she already knew. Searching for paths to the constructors that were *not* through the factory was difficult.

This question is an example of a *reachability question*. A reachability question is a *search for statements* across *feasible control flow paths* through a program for statements that match search criteria. Conventional search tools let developers use a variety of search criteria such as strings and statement attributes (e.g., field accesses, method calls) to locate matching statements globally in a program. Conventional code exploration tools such as slicing let developers traverse various types of paths through a program (e.g., dependencies, control flow graphs). Reachability questions fundamentally differ from both of these approaches by intersecting statements along paths with statements matching search criteria. A reachability question includes both a specification of paths *downstream* (after) or *upstream* (before) a set of statements and a set of search criteria describing statements to find along these paths. By specifying both paths and search criteria, reachability questions are much more specific, potentially resulting in a far smaller set of target statements found. Our studies indicate that many of the questions developers ask or should have asked are reachability questions containing both a specification of paths to search and search criteria describing statements for which to search [10].

We believe that a better understanding of how developers are *trying* to search their code, and the specific questions and strategies they have, can provide inspiration and guidance for novel types of tools that more effectively support answering these questions. During coding activities, developers refine high-level questions into lower level questions that they can answer using their tools or other strategies. At the highest level, developers ask factual questions about design decisions, such as, which design should I choose [9]? Developers often decompose these questions into searches for statements describing the code's *behavior* [10]. At the lowest level, developers ask questions about individual elements or relationships and explore the code around those elements [11]. To answer these questions, developers use today's tools to read code snippets, perform conventional searches, and traverse paths of method calls.

Tools targeting higher-level questions that developers ask have the potential to make a large impact by replacing the low-level actions developers must do now. While the highest-level questions are inherently conceptual and will require decomposition by a developer to answer, middle-level questions are often about searching for statements in a program. These questions are currently hard to answer because tools do not directly support them.

Several important design decisions must be made in designing tools for exploring paths through programs. All such tools explicitly or implicitly generate a graph of statements in a program. For example, a static slicer (e.g., [1]) builds edges between statements when statement *a* may *influence* statement *b*. In contrast, our proposed technique, which we call *static traces*, builds edges when statement *b* could feasibly follow *a* in some execution. This decision determines the paths through the statement graph that can be searched or traversed. Tools must also decide the types of searches that can be performed and how the results are depicted.

In this paper, we first describe how developers translate many common questions into searches across paths. We then discuss three important considerations in designing tools to help developers more effectively answer these questions.

2. DEVELOPER ACTIVITIES

Developers often ask reachability questions in the context of two different activities. As developers *propose* changes and consider their *implications*, they often wonder if something might break. When *debugging*, developers often search from a known method before a fault across downstream paths for statements causing the bug to occur. We [10] and others [6][11] have looked at the questions developers ask during these activities. An interesting finding is that developers spend much of these activities searching for statements across paths [10]. In the following sections, we describe how developers make use of both a specification of paths to search and search criteria specifying statements to find.

2.1 Debugging questions

Another friend tried to debug a bug involving both code he wrote and code included in his project from a framework. For some reason, his code computed incorrect results. After narrowing the problem down to a single Java Collection object, which contained the incorrect values, he tried to understand where the collection was being accessed and how it was being used. But, control flowed back and forth between his code and the framework, and finding the correct corresponding method in each case was both tedious and made keeping track of the path challenging. He wished to search downstream across this path for places where the collection was being mutated.

Developers often begin debugging tasks with a method in code that they believe immediately precedes the bug and try to search for statements causing the bug to occur. Much of the debugging process consists of locating these statements. In some situations, they read the code and use source browsing tools to traverse across calls. In other situations, developers guess the location of statements that might be relevant to the bug and use print statements or breakpoints to check if they are executed. Bugs can be challenging to debug when developers have difficulty locating relevant statements because they are located far away or when developers cannot determine which method calls to traverse.

Examples of debugging activities [10] suggest both that developers usually have search criteria in mind for the statements they are seeking, and that developers are trying to search over control flow paths from a particular starting location. For example, in our studies, a developer debugging a deadlock searched for statements acquiring resources. Another developer began at a method he knew corresponded to a command that just executed and searched downstream for the statement which caused an error to be generated to try to understand what was causing the error.

2.2 Implication questions

Developers also search when trying to determine the implications of a proposed change to the code. Before committing the time to implement a change, developers try to determine if it will work [6][9][10][11]. To answer this question, developers often attempt to understand how the code works now to determine how it would work differently after the change. We have seen that many of the strategies used for this understanding involve searching for statements across control flow paths.

For example, in our lab study [10], developers tried to determine if they could remove a call to a method. In order to understand what the call did, developers searched for side effects and calls into a framework that occurred downstream from the call. As existing tools do not directly support this kind of search (there are

Question	**Related downstream search**
What parts of this data structure are accessed in this code?	Search downstream for accesses to the data structure
What parts of this data structure are modified by this code?	Search downstream for writes to the data structure
What data is being modified in this code?	Search downstream for writes to any field
What exceptions or errors can this method generate?	Search downstream for throws or error calls
How do calls flow across process boundaries?	Search downstream for out of process messages
How is control getting from *a* to *b*?	Search downstream from *a* for *b*
What resources is this code using?	Search downstream for calls accessing or acquiring resources
What are the possible actual methods called by dynamic dispatch here?	Search downstream across feasible paths
How does this code interact with libraries?	Search downstream for calls to libraries
What is the difference between these similar parts of the code (e.g., between sets of methods)?	Compare statements downstream from each method
Question	**Related upstream search**
Is this tested?	Search upstream for unit test methods
What threads can reach this code or data structure?	Search upstream for thread creation calls
What is the “correct” way to use or access this data structure?	Search upstream for paths along which data structure is used
What is responsible for updating this field?	Compare paths along which field is written
What in this structure distinguishes these cases?	Search for reads from the structure upstream from each case
In what situations or user scenarios is this called?	Search upstream for framework callbacks denoting user actions
When during the execution is this method called?	Search upstream for framework callbacks or main methods
What parameter values does each situation pass to this method?	Search upstream for values which flow into parameters
Is this method or code path called frequently, or is it dead?	Search upstream from method or code path

Table 1. Questions developers ask can be answered by searching downstream or upstream along control flow paths.

no tools which can search for all calls into a framework), developers were only able to read snippets of code and guess which calls to traverse to find target statements. Unfortunately, the methods which might have answered these questions were located several calls away. As a result, all the developers gave up and did not find them, resulting in incorrect changes.

2.3 Further Examples

Our studies provide strong evidence that searching across paths is an important and pervasive part of coding activities. In a lab study of coding tasks [10], half of the bugs were related to searches over paths developers attempted or should have attempted. In observations of developers at work on their own tasks [10], 9 out of 10 of the longest debugging and implication investigations involved answering a question related to a single search that took tens of minutes to answer. Furthermore, many of the questions observed in other researchers' studies of developers are related to searches across paths. Table 1 shows how questions from two studies (an unpublished survey of developers and Silito's [11]) can be translated into searches across paths.

3. SEARCHING ACROSS PATHS

Given the evidence that developers need to search across control flow paths, we have begun designing a tool to make these searches easier. In doing so, we have explored three important design choices: (1) what kinds of paths will be searched over, (2) what can be searched for, and (3) how the results are displayed.

3.1 Paths

Previous systems have supported searching or traversing across various kinds of paths. A static slice includes edges between statements that may be control or data dependent in any execution [1], while a dynamic slice finds dependencies in a specific execution [7]. Other tools search across control flow graphs containing edges between statements that might possibly execute sequentially [5] or concrete traces describing the path in a particular execution [2]. Impact analysis tools often extend control flow graphs with additional edges for relationships such as type references [4].

However, we believe that the searches we have observed developers attempt, as described above, are not effectively supported by any of these approaches because developers wish to search across feasible control flow paths. While using static slices without traversal or search has traditionally been criticized for containing too much of the program (~30% on average [3]), static slices may sometimes also contain too little of the program While developers usually wanted to search for everything that might execute after (or before) a statement, slices only contain the subset of these statements that are control or data dependent.

While control flow graphs, by definition, include all possible control flow paths through a program, they also include infeasible paths that may never execute. In practice, this is an important issue in conventional GUI programs since they contain many flags and often use dynamic dispatch. For example, participants in one of our lab studies wished to search across paths through a bus. Many methods sent a message onto the bus, and many methods received messages, but only a small number of methods sent or listened to the specific messages of interest. In another example, widely implemented interfaces (e.g., COM interfaces in C++ or IRunnable in Java) on which a method is invoked results in control flow paths including many implementations that are never connected at runtime. Infeasible paths create false positives that may overwhelm the true positives.

In some debugging activities, traversing dynamic slices can be substantially more effective than using a debugger [8]. But, dynamic analyses do not let developers search over *all* behavior that might occur, which is particularly important in many implication investigations where developers need to understand all cases. And in some debugging situations, developers may work with long-running operations, stack dumps from the field where no reproduction steps are known, or bugs requiring specialized configurations or hardware. In these cases, it is advantageous to debug without running the program.

To address the limitations of existing types of paths, we are designing a new type of path – the *static trace.* Unlike dependency paths built by slicers, static traces contain all statements that may execute after (or before). Unlike both control flow graphs and static slicers, a static trace is an execution trace, containing methods multiple times in different execution contexts like a dynamic trace. And most importantly, static traces do not contain infeasible paths that a static analysis is able to show will never execute. But, unlike dynamic traces, static traces show behavior from all executions rather than a single execution.

3.2 What Can Be Searched For

Previous systems have let users search for strings in method names [2][4] or method text [5]. Our studies indicate that developers do indeed search by string for methods or for reads or writes to fields contained in specific types. In addition, developers wished to search by string and statement attributes (e.g., all calls into a framework, all reads to a field) or just by statement attributes, neither of which are currently supported. Statement attribute searches are very different from string searches in that developers do not yet have specific behavior in mind for which they can produce strings but wish to find all behavior of a specific type. Developers also sometimes wished to perform *nested searches* to compare or search across results produced by other searches. For example, developers wished to compare statements downstream from different methods or compare reads to a structure along alternative paths to understand the differences.

In our system, we propose to let users search by strings and statement attributes and to use the results produced by one search in subsequent searches and comparisons. We are also exploring interactions for quickly composing and refining searches.

3.3 Depicting the Results

Previous systems have presented users with either a list of target statements [4][5] or the path by which the target is reached [2]. We believe presenting targets without the path is often ineffective in situations such as when developers: (1) do not understand how the target was reached, (2) want to understand why the call occurred, (3) wish to compare paths by which the target is reached, (4) are interested in ordering relationships or connections between multiple targets, (5) wish to see the circumstances (variable values) necessary for a path to be taken, or (6) are skeptical that a path ever executes. Existing approaches for depicting paths are based on UML sequence diagrams which are designed for displaying paths in a single dynamic trace. While it is possible to express alternatives, the diagrams can quickly become unwieldy.

We are currently designing a new visual notation for describing control flow paths. Unlike a call stack, multiple callees of a method may be shown. Unlike existing notations for dynamic traces, our notation includes information about mutually exclusive calls and calls that may or may not execute and supports comparing paths in alternative situations. In order to remain compact, our

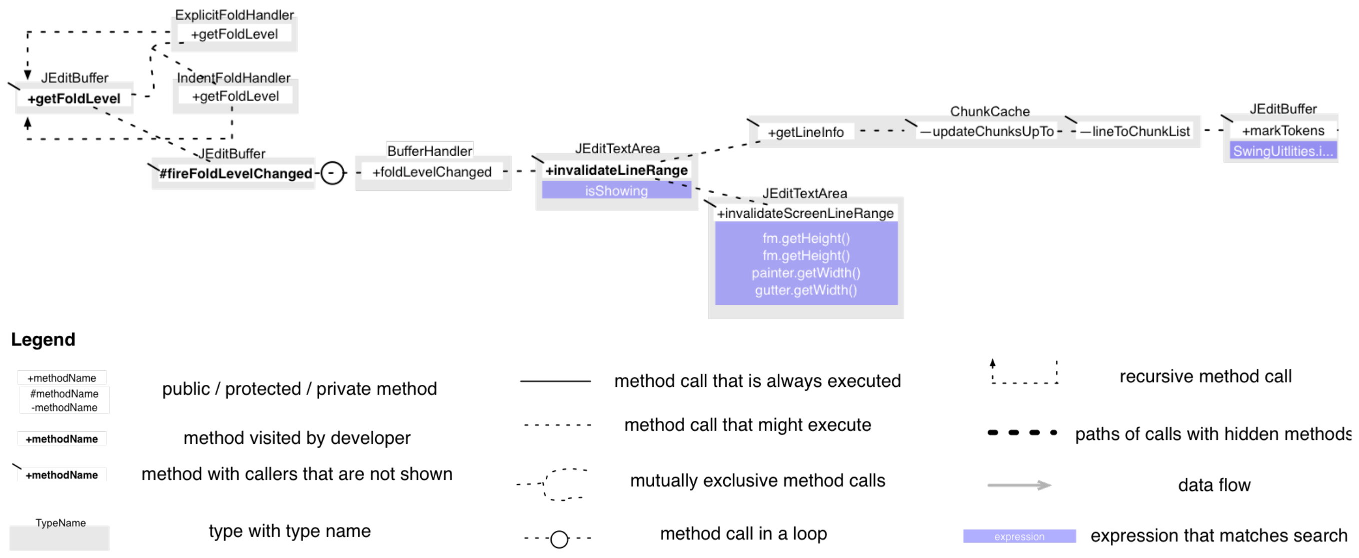

Figure 1. A mockup of our path visualization. Developers in one of our studies wondered why a call to JEditBuffer.getFoldLevel was necessary even though the return value was ignored. Maybe the method has effects by mutating fields or communicating with the framework? Some developers looked downstream for this behavior, but the relevant statements were several calls away, and developers failed to locate them. In the mockup, target statements (calls into the framework) are shown with a blue background and paths from the search origin (getFoldLevel) are depicted using a variety of visual attributes.

visualization combines methods occurring multiple times in recursive calls or when the call site is located in a loop. An important focus has been finding the right balance between displaying too much and not enough information. Only methods containing statements that matched a search are shown by default. Paths between these methods are shown with a single dashed edge that can be expanded to see the complete path. And we have considered several alternative levels of detail provided in the visual attributes shown. See Figure 1 for a mockup.

4. CONCLUSIONS AND FUTURE WORK

Recent studies indicate that searching across control flow paths is a widely used approach for answering many questions. Therefore, we are currently designing a tool to make this easier. But there are several challenges to creating such a tool. A static analysis is necessary to eliminate infeasible paths, but must be fast enough to compute results in response to user searches. The visualization should help users make sense of the paths without displaying an overwhelming amount of irrelevant information. We believe that better tool support for searching across control flow paths will help make many common coding activities easier, faster, and less error prone.

5. ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation, under grant CCF-0811610. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

6. 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., Myers, D., Storey, M. German, D. M., Oullet, D., Solois, M., and Charland, P. (2008). A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. In *Journal of Software Maintenance and Evoluation*, 20 (4), 291-315.

[3] Binkley, D., Gold, N., and Harman, M. (2007). An empirical study of static program slice size. In *TOSEM*, 16(2).

[4] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V. (2005). JRipples: a tool for program comprehension during incremental change. In *Proc. of the 13th Int. Workshop on Program Comprehension (IWPC).*

[5] Hill, E., Pollock, L., and Vijay-Shanker, A.K. (2007). Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. ASE.*

[6] Ko, A. J. DeLine, R., Venolia, G. (2007). Information Needs in Collocated Software Development Teams. In *Proc. ICSE.*

[7] Ko, A.J., and Myers, B.A. (2008). Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. ICSE,* 301-310.

[8] 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.

[9] LaToza, T.D., Garlan, D., Herbsleb, J.D., and Myers, B.A. (2007). Program comprehension as fact finding. In *Proc. ESEC/FSE.*

[10] LaToza, T.D., and Myers, B.A. (2010). Developers ask reachability questions. To appear in *Proc. ICSE.*

[11] 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).