IMPROVING FEATURE LOCATION BY COMBINING DYNAMIC ANALYSIS AND STATIC INFERENCES

Meghan Revelle
Advisor: David Coppit
Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185
meghan@cs.wm.edu, david@coppit.org

Abstract

Identifying the code needed to perform software maintenance and evolution tasks can be very difficult and time consuming for large, complex software systems. A number of techniques have been proposed that employ either static or dynamic analysis to isolate code related to a feature of interest. Static approaches suffer from poor precision by including irrelevant code, while dynamic approaches can suffer from poor recall by excluding relevant code. This paper presents and evaluates a hybrid approach for feature location that augments execution trace analysis with an analysis based on the program structure and associated semantics. This approach improves the completeness of feature location results by expanding the mapping of features to code provided by dynamic analysis in light of the static structure and semantics of the program. The approach is evaluated relative to a well-known dynamic approach using several features in an open source system.

1. Introduction

Software maintenance and evolution tasks first requires programmers to understand the implementation of an existing software system. To do so requires identifying the parts of the source code that are relevant to specific functional requirements, an activity known as feature location [26]. Once adequate comprehension has been achieved and the code corresponding to a feature or task is found, the actual modification of the software can be performed. The first step of gaining knowledge of the software and identifying important code often takes more time than the second step of modifying it [13, 14]. With maintenance dominating the software life cycle [3], reducing time spent on understanding and searching the software for code has the potential to greatly improve productivity.

Automated feature location tools that aid programmers in exploring and comprehending large software systems fill a critical need. Most automatic feature location approaches focus on one type of analysis, either dynamic or static. The dynamic approaches collect and analyze execution traces, and their quality is tied to the quality of the test cases used to execute the software, sometimes resulting in an incomplete picture of the code implementing a feature. On the other hand, static approaches do not execute the software and tend to return too much code, not all of it relevant to the feature of interest. Combining the advantages of dynamic and static analysis can minimize their disadvantages and yield better results.

This paper introduces a new hybrid feature location technique. The proposed approach is based on the idea that dynamic analysis cannot provide information about code that was not executed, therefore, the features implemented by unexecuted code must be determined statically, if possible. The features that correspond to unexecuted code are inferred by considering static program structure and programming language semantics. Execution traces are collected, and the traces are statically expanded to include unexecuted code that is potentially relevant to a feature. Then, the expanded traces are subjected to formal concept analysis to determine the mapping of features to code.

This paper makes the following contributions:

• A semi-automatic feature location technique that combines dynamic and static information to make inferences about the features implemented by unexecuted code.

• A quantitative evaluation of the accuracy and completeness of this new feature location technique.

Experimental results demonstrate that this new hybrid feature location technique is effective at identifying more of
the code corresponding to features than a wholly dynamic approach.

The next section describes existing methods of feature location and program exploration. Section 3 presents an example motivating the need for a hybrid feature location technique that incorporates dynamic analysis and static structural and semantic information. In Section 4, this new approach is explained in detail. Section 5 shares the results of experimental evaluation of the approach, and Section 6 concludes.

2. Related work

Since feature location is an important part of software maintenance and evolution, many techniques for it already exist. This section reviews some of these existing approaches by categorizing them as either static, dynamic, or hybrid feature location techniques.

2.1. Static feature location

Most static feature location techniques are either structural or lexical. Structural approaches such as Abstract System Dependence Graphs (ASDG) [5], Concern Graphs [21], and Suade [23] are used to navigate dependencies. ASDGs and Concern Graphs model structural dependencies of a program, and a programmer manually navigates the ASDG or Concern Graph to find relevant methods. Suade automatically determines suggestions for investigation based on structural relations in the code. These three approaches require a seed method, so they are more suited to programmers who already have some knowledge of the software they are investigating.

Static, lexical approaches use comments and identifiers to locate code relevant to a feature. Most simplistically, grep can be used to search a code base using regular expressions. More sophisticated approaches utilize information transparency [9], information retrieval (IR) [19], or natural language [22]. The drawback of tools that rely on static information, specifically lexical information, is that they are best applied to systems that adhere to a naming convention. These approaches are less effective for systems with poor identifier names.

A number of tools use both structural and lexical information to locate pertinent code [10, 11, 12, 29]. These tools work by using lexical information to prune irrelevant structural relationships, or vice versa. Some tools, like SNIAFL, rely on IR techniques, while others like Dora use a method relevance score to determine germane code.

Program slicing [24] is a type of analysis used to identify the parts of a system that may affect the value of a variable within a program. Static slicing returns all possible program elements that could impact the point of interest. Thus static program slicing is a primitive form of feature location if the point of interest corresponds to a feature. Static slicing is prone to being expensive to calculate and returning very large slices, which limits its usefulness for feature location.

One final area of static feature location is repository mining [28, 30]. By using data mining techniques, software repositories can be mined to discover program elements that are frequently changed together. Thus, code that implements a feature can be found based on change history. Repository mining approaches require a large history for analysis, and if the feature of interest is not changed often, the approach is less effective.

2.2. Dynamic feature location

Some of the earliest work on mapping features to code was software reconnaissance [25, 26]. It is a fully dynamic approach that compares an execution trace of a program when a feature is invoked to a trace of the program when the feature is not executed. The code that appears only in the trace that invokes the feature is assumed to be related to that feature. Software reconnaissance has recently been expanded and improved by adding new criteria for selecting execution scenarios for collecting traces and analyzing those traces differently using rankings [7].

Like static slicing, dynamic program slicing [1] can be used to locate the implementation of a feature [15, 16]. Wong et al. [27] developed a more sophisticated slicing approach. Their execution-slice based technique identifies the code unique to a feature and common to a group of features. Sets of tests that invoke and exclude a particular feature are carefully selected and then executed, and the resulting execution slices are compared.

Dynamic feature location approaches have a number of disadvantages. The quality of the results given by these techniques is tied to the availability and quality of test cases. If the test cases used to collect traces only partially exercise a feature, important code may be missing from the results. Also, these techniques are best suited to identifying code for features that can be invoked at the user level.

2.3. Hybrid feature location

Hybrid methods of feature location seek to leverage the benefits provided by both static and dynamic analysis. Eisenbarth et al. [6] developed a technique for feature location that is mostly dynamic but includes some static analysis. They apply formal concept analysis to execution traces to produce a mapping of features to the program’s methods. By inspecting a static dependency graph, a programmer can refine the mappings. Koschke and Quante [17] expanded this approach to do more fine-grained analysis at the basic
block level. The dynamic portion of this approach that uses formal concept analysis (FCA) on execution traces is an automatic means of feature location. However, the static portion of inspecting dependencies is manual. The approach presented in this paper seeks to automate the inclusion of static analysis.

Instead of applying FCA to execution traces, SITIR [18] and PROMESIR [20] use Latent Semantic Indexing (LSI), an IR technique that indexes textual information in source code so that it can be queried. In SITIR, a single execution trace can be filtered using LSI to extract all the code relevant to the feature of interest. In PROMESIR, LSI is combined with a dynamic analysis technique known as SPR [2] to give a ranking of methods likely relevant to a feature. Both techniques focus on ranking methods based on their relevance to a feature and not necessarily on identifying all code corresponding to that feature.

### 3. A motivating example

This section presents an example motivating why combining dynamic and static analysis is important for automated feature location. This section also introduces the notion of static inferencing, statically inferring the features implemented by a portion of code given some dynamic information.

For the example in this section, consider GNU sort\(^1\), an open source program that sorts lines of text files. This program has many command line flags that control how files should be sorted, and each of these flags can be considered a feature. For example, the `-n` flag corresponds to the feature for sorting string numerical values. The test suite that comes with the distribution does not test every feature, and of the features it does test, the test suite does not necessarily test them completely. As a result, some code is not executed at all. The question is, was that code not executed because it pertains to a feature that was not tested, or was that code not executed because a more comprehensive test suite is needed to obtain full coverage of a feature that was tested?

Running GNU sort's test suite and collecting code coverage traces reveals that the method `fraccompare()` is never executed. This method compares strings containing decimal fractions less than one, so it seems relevant to the feature for sorting numerical values invoked by the `-n` flag. It was not executed because none of the test cases for the numerical sort feature included decimal fractions as input. A purely dynamic feature location technique using this test suite would not identify `fraccompare()` as being relevant to the numerical sort feature. This situation can be avoided by combining the information from dynamic analysis with static structural relationships. The `fraccompare()` method is only called from within the `numcompare()` method, which is only executed when the `-n` flag is invoked. Since the collected execution traces show `numcompare()` to be exclusive to the feature for sorting numerical values and it can statically be determined that `fraccompare()` is only reachable from `numcompare()`, it can be inferred that `fraccompare()` is also related to the same feature. Additionally, any unexecuted code in `numcompare()` can be inferred to be relevant to the numerical sorting feature. These are examples of static inferences.

As a proof of concept of the usefulness of static inferencing, they were manually applied to sort. The next section more fully explains the different types of static inferences applied. To evaluate feature location with static inferences, it was added to the dynamic portion of the well-known feature location technique of Eisenbarth et al. [6] that uses execution traces and formal concept analysis. The results of the purely dynamic feature location are compared to the same approach with the addition of static inferences. The comparison is in terms of the information retrieval metrics of precision and recall. Precision is the fraction of code identified that is actually relevant to a certain feature. Recall is the fraction of code relevant to a certain feature that was actually identified. To use these metrics requires an established set of code known to be relevant to the features. For the relevant sets, the lines of code identified for each feature of sort in [4] are used.

Table 1 shows the results for precision and recall using the purely dynamic approach and the same dynamic approach in conjunction with static inferencing. The precision and recall results are poor for the purely dynamic analysis.
for two reasons. First sort’s test suite is not comprehensive. Second, the relevant sets include non-executable code such as comments and whitespace which are not included in execution traces. However, with the addition of static inferencing, the precision and recall results improve significantly in most cases. This small example reveals that static inferencing can improve dynamic feature location by compensating for an incomplete set of test cases without additional effort on the part of the programmer.

4. Feature location with static inferences

This section describes feature location with static inferences and how programmers can use it during maintenance to locate relevant feature code. The main contribution of the approach is the automatic identification of code corresponding to a feature by expanding dynamic execution traces with information inferred from static relationships. The following steps give the details of how the technique achieves its intended purpose, and Figure 1 summarizes the approach.

Step 1. Identify Features. To use feature location with static inferences, programmers should begin by determining the features of the system they are interested in locating code for and how those features are invoked, such as through a command line flag, graphical interface button, or particular input value. For maintenance or evolution, generally a bug report or change request can be used as the starting point for determining features of interest. Ideally, existing test cases that invoke the features of interest will be available for use in the next step. If no test cases are at hand, programmers will have to create a small test suite of their own. Even if the programmer-created tests poorly exercise the features, the later step in the process that incorporates static information can help compensate for this fact.

Step 2. Collect Execution Traces. To collect execution traces, the software should be executed using the test cases of the previous step and a code coverage tool. The code coverage tool should present line-level reports of the code executed for each test. In the case studies presented in Section 5, EMMA\textsuperscript{2} was used to collect execution traces.

Step 3. Static Inferencing. The reports given by code coverage tools do not accurately reflect all of the code involved in the implementation of a feature in two ways. First, execution traces do not include non-executable code such as brackets, variable declarations, and comments because of limitations in code coverage tools. Second, when a test suite does not include a test for a special case of a feature, part of the feature’s code was not executed. A trace profile is a revised version of an execution trace that adds excluded code and removes unnecessarily included code according to static inferencing. There are two types of static inferencing: semantic and structural. Semantic static inferencing takes care of the limitations in code coverage reporting, while structural inferences expand execution traces to include unexecuted code. The semantic static inferences are as follows:

- Field and variable declarations are included in a trace profile if any executed statement references them.
- Access specifiers (public, private, protected, etc.) are included in a trace profile if any statement in the enclosing structure was executed.
- The conditional portion of if statements are excluded from a trace profile unless their clauses were executed or they fall through to another executed case.
- Loops are removed from a trace profile if the body of the loop was not executed.
- Brackets enclosing blocks of statements are included in a trace profile if any statement in the block was executed.
- Cases of switch statements are excluded from a trace profile unless their clauses were executed or they fall through to another executed case.
- Method headers are included in a trace profile if any statement within the method was executed. Otherwise, the entire method is excluded.
- Comments or whitespace within a block are included in a trace profile if a statement within the enclosing block was executed.
- Comments or whitespace before a method or class header are included in a trace profile if any statement within the method or class was executed.

Table 2 shows the application of several of these semantic static inferences to a small example program. Method and class headers as well as brackets, variable declarations

\textsuperscript{2}http://emma.sourceforge.net/
and comments have been added to the trace profiles. Notice that for the $A = B$ trace profile, the while loop condition has been removed because the body was not executed. The inferences help to more fully flesh out the implementation of features and better enable FCA to determine the correspondence of features to code.

While the first type of static inferences deals with language semantics, another type of static inferencing involves structural relationships. Static structural inferences are used to expand execution traces to include code that was not executed due to poor test case quality but still is relevant to a feature. The structural static inferences are as follows:

- If unexecuted code exists in a method and all other code in that method is related to a single feature, infer that the unexecuted code is also related to that feature.
- If a whole method was not executed, determine the callers of the method. If a caller of the method is known to be related to a feature, infer that the unexecuted method is also relevant to the feature.
- If any calls to a method known to be related to a feature are unexecuted, infer that the unexecuted calls and the basic blocks that contains them are related to the feature as well.

**Step 4. Perform Formal Concept Analysis.** The trace profiles can be organized into a table called a formal context, the input for formal concept analysis (FCA) [8]. Each row of the formal context corresponds to a line of code in the program. Each column corresponds to an executed test case, labeled in terms of the features executed by the test. FCA mathematically identifies maximal groups of elements (lines of code) with a common set of properties (features). Sets of elements with shared properties are called concepts, and given the sets defined by the concepts, it is possible to define a partial order among them to create a concept lattice. Figure 2 is a concept lattice for the trace profiles in Table 3. For readability reasons, lines of code have been replaced with numbered elements and correspond to the labels below a node in the lattice. Labels above a node in the lattice indicate the properties (features) of that concept. For this example, the three test cases executed feature $F_1$, feature $F_2$, and both features $F_1$ and $F_2$, respectively.

**Step 5. Interpret the Concept Lattice.** Concept lattices reveal the mapping of features to code and the relationship between features. Concept lattices for large software systems can be too complex for a human to interpret. Fortunately, a concept lattice can be automatically interpreted and traversed for feature location, as described in [6]. Concepts with only one feature property reveal the code solely
relevant to that feature. For example the lattice in Figure 2 shows that element $e_6$ is relevant to feature $F_1$. If a feature was executed by several tests, more code that is specific to that feature can be found in the infimum, the closest common concept toward the bottom of the lattice which can be reached by starting at concepts that have the feature in their label. In Figure 2, the concept with element $e_2$ is the infimum of its two parents. Therefore, both element $e_6$ and $e_2$ are relevant to the feature $F_1$ since $e_2$ was executed by all tests involving feature $F_1$.

One goal of developing feature location with static inferences is to keep the programmer’s role as small as possible. Currently, the programmer’s role in the process of feature location with static inferences is minimal, requiring only determining the features executed by existing test cases or creating a small number of new tests. Another goal is to accurately find all the code relevant to a feature. The next section explores how well feature location with static inferences lives up to that goal.

5. Experimental evaluation

In order to evaluate the performance of using feature location with static inferences, a case study was performed on jEdit\(^3\). In the study, both dynamic feature location and the new hybrid feature location approach with static inferences were used to locate the lines of code the implement features.

5.1. Experiment design

This subsection describes the subject system, variables, measures, and methodology used in the case study to evaluate hybrid feature location with static inferences.

5.1.1. Subject system

jEdit version 4.2 is an open source programmer’s text editor written in Java. It consists of nearly 87,000 lines of source code, or over 135,000 lines of code including internal comments. jEdit has just under 700 classes and 4,500 methods. It has many features, but only three are considered in this study. They are “Incremental Search,” “Hyper Search,” and “Go To Marker.” The incremental search feature allows a user to search for a specified pattern in the current buffer. Search results are highlighted one at a time, and the user can navigate to the next or previous match. In addition to searching for patterns in the open buffer, the hyper search feature also allows searching in all open buffers and in a directory. Search results are presented list form in a separate window. The final feature considered, “Go To Marker,” allows a user to go directly to a marker, a location in an open buffer previously set by the user.

5.1.2. Variables and measures

The independent variable in this study is the feature location technique used, either a purely dynamic approach or feature location with static inferences. The dynamic technique used is a slight modification of the approach described by Eisenbarth et al. in [6] that uses formal concept analysis of execution traces. The difference is instead of collecting traces at the method level, traces are collected at the line level in this study for more fine-grained results. The other technique considered is feature location with static inferences as presented in Section 4.

The dependent variable in this study is the effectiveness of each feature location technique, measured in terms of the information retrieval metrics of precision and recall. Precision is the fraction of lines of code reported by a technique that are relevant. It is calculated by dividing the number of

---

\(^3\)http://www.jedit.org/
Table 4. Descriptions of the test cases used to locate features in jEdit.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Test Cases</th>
</tr>
</thead>
</table>
| Incremental Search | 1. Use the search menu item, toolbar button, and keyboard shortcut to search the open buffer for the first occurrence of some text.  
          | 2. Use the search menu items to search for some text, find the next occurrence, and the previous occurrence.  
          | 3. Use the search menu item to search for a regular expression.  
          | 4. Use the search menu item to search for some text, ignoring case.  
          | 5. Use the search menu item to search for some text using autowrap.  
          | 6. Use the search menu item to search backwards for some text.  
          | 7. Use the search tool bar to search for some text.  
          | 8. Use the search for some text that is not in the open buffer. |
| Hyper Search | 1. Use the hyper search menu item and keyboard shortcut to hyper search for some text.  
          | 2. Use the hyper search menu item to search for a regular expression.  
          | 3. Use the hyper search menu item to search for some text, ignoring case.  
          | 4. Use the keyboard shortcut to hyper search in a directory.  
          | 5. Use the keyboard shortcut to use the search toolbar to hyper search for some text.  
          | 6. Use the hyper search menu item to search for some text that is not in the current buffer. |
| Go to Marker | 1. Use the marker menu item and keyboard shortcut to go to a marker.  
          | 2. Use the go to marker menu item and keyboard shortcut to try to go to an unknown marker.  
          | 3. Use the command line option “+marker” to go to a marker when jEdit starts.  
          | 4. Use the marker menu items and keyboard shortcuts to go to the next and previous markers. |

reported relevant lines of code by the total number of lines of code reported. When precision is high, it means a feature location technique found few irrelevant lines of code. Recall is the fraction of relevant lines of code reported by a technique. Recall is calculated by dividing the number of relevant lines of code reported by the total number of relevant lines of code. A high recall value means a feature location technique missed few relevant lines of code.

5.1.3. Methodology

In this study, the objective is to identify the code that implements three features in jEdit and determine if using static inferences results in more relevant code being identified than a purely dynamic approach despite an incomplete set of test cases for a feature. To do so, the process in Section 4 was followed. A set of test cases was developed for each feature by examining jEdit’s user interface. These test cases are described in Table 4. Code coverage was collected for each of these test cases individually. Additionally, code coverage was also generated for the execution of jEdit with multiple test cases. Five groups of two, three, and four test cases were randomly selected for each feature. These groups represent a more robust but still incomplete set of tests. For the feature location with static inferencing, the execution traces were subjected to static inferencing before performing FCA, while only FCA was used for the purely dynamic technique.

The ideal way to evaluate any method of feature location is to compare its results to the code known to implement a certain feature. However, it is rare that such information about the mapping of features to code is available. Since this information is not known for jEdit, another standard for relevant feature code is needed. In this study, the relevant code for a feature is determined by collecting execution traces of the full set of test cases that execute that feature and performing FCA on those traces.

In order to get a clear comparison of the results of the dynamic approach versus our hybrid one, only the structural static inferences were used in this study. Since the code deemed relevant to a feature was determined by execution traces, non-executable code such as comments and whitespace are not included. Semantic static inferences add non-executable code to the execution traces. Therefore, semantic static inferences were disabled in this study so that the results of the two feature location techniques are more comparable.

5.2. Results

Figure 3 shows graphs of precision and recall for both feature location techniques examined in this study. Each bar represents the average precision or recall of five executions of jEdit using the test cases in Table 4. For each execution, a random selection of test cases was used. In the case of only one test case, a single test case for each feature was picked at random. When multiple test cases were used, the appro-
5.3. Discussion

Feature location with static inferences can be used by programmers performing maintenance or evolution on an existing software system with which they are unfamiliar. They may develop some test cases for a few features of interest by consulting any available documentation or the system’s user interface. Without examining the source code, it is likely that their set of test cases does not exercise every aspect of the features. However, if they are using feature location with static inferences, they do not need to worry about creating a complete set of test cases for the features of interest. This study found that feature location with static inferences helps identify code relevant to the implementation of a feature, even when some of that code was not executed. Static inferencing in part compensates for an incomplete set of test cases.

Feature location with static inferences outperformed purely dynamic feature location in terms of recall, but it needs further development to improve its precision. The static inferences identify too much irrelevant code, meaning programmers might believe some lines of code contribute to a feature when they do not. For future work, the precision of our technique needs to be improved to filter out the excess irrelevant code that is reported.

5.4. Threats to validity

This section discusses some of the issues that may have affected the results of this study. These issues potentially limit the conclusions that can be drawn for the study and the ability to generalize the results.

The first threat to validity is the code that is relevant to a feature is subjective. For this study, a feature’s relevant code was not determined manually but by using dynamic feature location using a robust set of test cases. However, the relevant code may not be an accurate representation of a feature, meaning our conclusions may have weak validity. All forms of dynamic program analysis suffer from the limita-
tion that their results are only as good as the test cases used to execute a system. This threat to validity was minimized by carefully developing the test cases for each feature.

Another threat is the fact that the test cases used to determine the relevant code were created by the authors who do not have expert knowledge of jEdit. Therefore, the test cases may poorly exercise the chosen features. Using poor test cases may be sufficient for evaluating feature location with static inferences, but using bad test cases to determine relevant feature code would weaken the results reported by this study. Again, this threat was minimized by carefully developing a full set of test cases developed for each feature.

Finally, since this study only investigated three features in a single system, the reported results may not generalize to all software systems. Further evaluation on other systems will need to be performed to determine the extent to which the results can be generalized.

6. Conclusions

To aid programmers in easily identifying relevant feature code, this paper has presented a feature location technique that augments dynamic information by statically inferring additional relevant code. The approach does not require programmers to know much about a specific software system to be used. Experimental results indicate that our new hybrid feature location approach is able to use static inferences to expand the information from dynamic analysis to include more relevant feature code that was not executed. Currently, the benefits of our technique come with a tradeoff. The static inferences add irrelevant code to the feature location results. Future work will focus on reducing the amount of irrelevant feature code reported by our approach in order to achieve accurate and complete results.

7. Acknowledgments

This research is supported by the Virginia Space Grant Consortium and a grant from the Air Force Office of Scientific Research (FA9550-07-1-0030).

References


