An Exploratory Study on Assessing Feature Location Techniques

Meghan Revelle and Denys Poshyvanyk
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
meghan@cs.wm.edu, denys@cs.wm.edu

Abstract

This paper presents an exploratory study of ten feature location techniques that use various combinations of textual, dynamic, and static analyses. Unlike previous studies, the approaches are evaluated in terms of finding multiple relevant methods, not just a single starting point of a feature's implementation. Additionally, a new way of applying textual analysis is introduced by which queries are automatically composed of the identifiers of a method known to be relevant to a feature. Our results show that this new type of query is just as effective as a query formulated by a human. We also provide insights into situations when certain feature location approaches work well and fall short. Our results and observations can be used to guide future research on feature location.

1. Introduction

Software maintenance and evolution tasks require programmers to understand specific parts of an existing software system [12] which necessitates locating the source code that implements functionality, an activity known as *concept assignment* [2] or *feature location* [21]. Most existing feature location techniques are effective at finding a starting point of a feature's implementation, i.e., one method that is relevant to a feature [13, 15, 16]. However, a single method is rarely the sole contributor to a feature. For feature location approaches to be truly effective, they need to find *near-complete* implementations of features. We define the term *near-complete* to mean a partial but close to total set of methods that implement a feature since knowing all the methods that implement a feature is rather subjective [18].

This paper presents an exploratory study of ten feature location techniques that use various combinations of textual, dynamic, and static analyses. The approaches are evaluated in terms of how well they locate near-complete implementations of several features in *jEdit* and *Eclipse*. As part of the assessment, we designed easy-to-follow evaluation guidelines. We also explored a new mechanism for automatically formulating queries for textual analysis.

Our results highlight the challenge of feature location since no single technique was universally successful. We provide observations of situations when the approaches work well and when they fall short. One promising result is that our new automatically created queries for textual analysis perform comparably to queries formed by a human. Overall, the results of this exploratory study can be used to improve the development of feature location to find near-complete implementations of features.

2. Feature location techniques

A *feature* is a functional requirement that produces an observable behavior which users can trigger [8]. Examples include spell checking in a word processor or drawing a shape in a paint program. The term feature is intentionally defined weakly in the literature so it is suitable in many situations [1, 7]. *Feature location* is the activity of identifying the source code elements (i.e., methods) that implement a feature [21]. We investigate several approaches to locate a feature's source code using textual, dynamic, and static analyses as well as their combinations.

2.1. Core techniques

Textual analysis. One approach to locate features is to determine textual similarities among a query and source code elements (e.g., methods) using an information retrieval technique known as Latent Semantic Indexing (LSI) [5]. Users can formulate queries in natural language (*nl-queries*) or from the identifiers and comments of a known relevant method (*method-queries*). LSI returns a list of all the methods in the software ranked by similarity to the query.

Dynamic Analysis. Another approach to feature location uses dynamic analysis [21]. To collect dynamic information, users execute *scenarios* that trigger a feature. A scenario is a sequence of user inputs to a system. As scenarios are being run, *traces* are collected. A trace is a list of events that occurred during execution. We focus only on method invocation events. There are two types of traces we consider. *Full traces* [21] capture all events from a system's start-up to shutdown. *Marked traces* [13, 19] only capture events during part of a system's execution such that users can start and stop tracing at will.

Static Analysis. Static analysis provides information on different types of dependencies in a system. We use light-weight static analysis focusing on method invocations in a static **p**rogram **d**ependency **g**raph (PDG) [4, 10, 17]. Using

JRipples[1][3], we obtain a PDG in which nodes are methods and edges are method invocations. Starting at a *seed method* that is relevant to a feature, other methods pertinent to that feature can be found by traversing the PDG.

2.2. Combined techniques

Textual Analysis. We consider textual analysis to be our baseline approach and evaluate it under two configurations: using *nl-queries* as in [13] and using our new *method-queries*. We call these approaches IR_{query} and IR_{seed}, referring to the fact that the textual analysis used is a form of information retrieval. The IR_{query} approach was introduced in [14], whereas IR_{seed} is new.

Textual Analysis plus Dynamic Analysis. To combine textual and dynamic information, methods that are not executed are removed from the ranked list provided by textual analysis. We investigate all configurations of queries and traces: IR_{query} + Dyn_{marked}, IR_{query} + Dyn_{full}, IR_{seed} + Dyn_{marked}, and IR_{seed} + Dyn_{full}, where "Dyn" stands for dynamic analysis and subscripts denote the type of trace. IR_{query} + Dyn_{marked} is like [13], while IR_{query} + Dyn_{full} is similar to [15]. The two other combinations are novel.

Textual, Dynamic, and Static Analyses. The final feature location techniques we evaluate incorporate all three types of information: IR_{query} + Dyn_{marked} + Static, IR_{query} + Dyn_{full} + Static, IR_{seed} + Dyn_{full} + Static, and IR_{seed} + Dyn_{full} + Static. The IR_{query} + Dyn_{full} + Static approach is conceptually similar to Cerberus [6], but instead of using prune-dependency analysis, it uses light-weight static analysis. The other three combinations are new.

Unlike when combining textual and dynamic analysis, static analysis does not involve pruning a ranked list. Instead, it entails exploring a PDG to find relevant methods and then ranking them. Searching begins at a seed method and expands to its static neighbors (parents and children). If its neighbors have a textual similarity above a threshold and were executed in a given scenario, exploration continues with the neighbors' neighbors. Once no more methods can be found that meet the criteria, searching stops and the list of results is sorted by textual similarity values.

In total, we investigate ten feature location techniques, many of which are novel because they involve *method-queries*. There are other combinations of textual, dynamic, and static analysis that we did not study. We decided against including these approaches since they do not produce a ranked list and the results of using standalone static and dynamic analyses are available elsewhere [4, 7].

3. Exploratory study

We performed an exploratory study to evaluate these ten feature location techniques. This section outlines our research goals, subject systems, and methodology.

[1] http://jripples.sourceforge.net/ (verified on 01/18/09)

3.1. Research questions

We set out to answer a number of research questions in this study. These **r**esearch **q**uestions (RQ) are:

- **RQ1**: What is the best combination of textual, dynamic, and static analyses for feature location? Specifically, which techniques are most effective at finding multiple feature-relevant methods?
- **RQ2**: Which type of IR query produces better results in terms of finding multiple methods associated with a feature, an *nl-query* provided by a user (e.g., requires human effort in formulating a query) or a *method-query* using the text of a seed method (completely automatic)?
- **RQ3**: Which type of execution trace, *marked* or *full*, discovers more methods that implement a feature?

3.2. Subject software systems

For our study, we chose two open-source Java systems of different sizes and from different domains. *jEdit*[2] is a highly configurable and customizable text editor. We used version 4.3pre16 which has approximately 105KLOC in 910 classes and 5,530 methods. We studied four features from *jEdit* chosen from fulfilled feature requests in the "Patches" section of its online tracking system.

- **Patch #1608486, *Support for "Thick" Caret*** adds a configurable option to make the cursor two pixels wide instead of one so it is easier to see (6 methods in patch).
- **Patch #1818140, *Edit History Text*** adds the ability to edit the history text of searches (5 methods in patch).
- **Patch #1923613, *Reverse Regex Search***, adds the ability to search backwards with regular expressions (2 methods in patch).
- **Patch #1849215, *Bracket Matching Enhancements***, adds the ability to match angle brackets (2 methods in patch).

Eclipse[3] is an integrated development environment. Version 2.1 consists of approximately 2.3MLOC in over 7K classes and 89K methods. With *Eclipse*, we chose to study fixed bugs corresponding to misbehaving features.

- **Bug #5138**[4] – Double-click-drag to select multiple words is broken (6 methods in patch).
- **Bug #31779**[5] **–** UnifiedTree should ensure file/folder exists (3 methods in patch).
- **Bug #19819**[6] – Add support for Emacs-style incremental search (19 methods in patch).
- **Bug #32712**[7] – Repeated error message when deleting and file is in use (6 methods in patch).

[2] http://www.jedit.org/ (verified on 09/18/08)
[3] http://www.eclipse.org/ (verified on 09/18/08)
[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=5138
[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=31779
[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=19819
[7] https://bugs.eclipse.org/bugs/show_bug.cgi?id=32712

3.3. Methodology

We briefly describe our methodology. Interested readers can find more details in an online appendix[8].

Textual Analysis. We formulated *nl-queries* by reviewing the description and comments in the thread for a patch/bug in *jEdit* and *Eclipse*'s issue tracking systems. Methods from the patches were randomly chosen to form *method-queries* for each feature.

Dynamic Analysis. We created one scenario per feature to collect traces. We devised *jEdit*'s scenarios from the description and comments for the patch. For *Eclipse*, two bug reports had steps to reproduce the errors which were used as the scenarios for those features. The scenario for bug #31779 is reused from [13]. For bug #19819, a scenario was created in which the behaviors of the feature, as described in the bug report, were exercised.

Static Analysis. The seed methods were the same methods used for constructing *method-queries*. As explained in Section 2.2, static analysis relies on a textual similarity threshold. We set the threshold using an adapted gap threshold technique [14, 22]. We incorporated a relaxation strategy: if the size of a ranked list did not reach our minimum of ten methods, we decreased the threshold by 0.05 and repeated the procedure again.

3.4. Relevancy assessment

We restrict our evaluation to the top ten methods generated by a feature location technique because other researchers have shown that users are generally unlikely to look at more than ten list elements [23]. If most of an approach's top ten methods are false positives, examining results lower in the list is unlikely to be worth the cost.

In reviewing the top ten methods of a technique, well-defined criteria are needed for judging whether they are relevant to a feature. While we knew the methods modified in the feature's patches, we did not use them as evaluation criteria because a bug may pertain to a subset of a feature's relevant methods. Instead, we adapted an approach used by Robillard et al. [18]. We presented programmers with lists of methods and asked them to determine the relevance of each method to a feature. The programmers were given source code, an executable, a description of a feature and how to invoke it, and the following guidelines on how to determine if a method is relevant to a feature or not.

1. Method names that are similar to the words in the feature's description are good indicators of relevant code, but the method's source code should be inspected to ensure the method is actually relevant to the feature.
2. Determine if the method is relevant to the feature by asking "*Would it be useful to know that this method is associated with the feature if I had to modify the feature in the future?*"
3. If most of the code in the method seems relevant to the feature, classify the method as *Relevant*. If some code within the method seems relevant but other code in the method is irrelevant to the feature, classify the method as *Somewhat Relevant*. If no code within the method seems relevant to the feature, classify it as *Not Relevant*.
4. If unable to classify the method by reviewing its code, explore the method's structural dependencies. If the method's dependencies seem relevant, the method probably is also relevant.

Having multiple programmers follow these guidelines and evaluating based on the agreement among them eliminates any one individual's bias.

4. Results

To evaluate the ten feature location techniques, one author classified every method in the ranked lists of all eight features without knowing which approach produced each list. To give support to the categorizations, we solicited volunteers to do the same for one feature (*jEdit*'s *thick caret)* and compared the results to the author's. The author's and the volunteers' results agreed over 90% of the time on the classification of relevant methods. The details of how agreement was computed, plus, additional results, are available in an online appendix[8]. The average percentage of relevant, somewhat relevant, and not relevant methods found in the top ten lists of each feature location technique are in Table 1. A discussion of the results is below.

4.1. Research question 1

For *jEdit*, the techniques that found the most relevant methods on average were IR_{query} + Dyn_{marked} and IR_{query} + Dyn_{marked} + Static. For *Eclipse*, there were three approaches that performed the best: IR_{query} + Dyn_{marked} + Static, IR_{query} + Dyn_{full} + Static, and IR_{seed} + Dyn_{marked} + Static. Different programmers may consider the somewhat relevant methods as part of a feature. If these methods are considered important, then IR_{query} + Dyn_{marked} is the best performing technique in the *jEdit* study and IR_{seed} + Dyn_{marked} + Static for *Eclipse*.

Since IR_{query} + Dyn_{marked} and IR_{query} + Dyn_{marked} + Static performed the same for *jEdit*, these results suggest that adding static analysis provides no additional benefits over a combination of only textual and dynamic analysis. Combining textual and dynamic analysis involves eliminating unexecuted methods from a ranked list, but using static analysis entails building a new list from scratch. Only methods with a static dependency to the seed are included. Therefore, methods that are located by a combined textual-dynamic approach may not be found by static analysis. However, the *Eclipse* results suggest that static analysis does aid feature location. In static analysis, if a method did not meet the textual similarity threshold, then exploration down that path of the PDG would halt. LSI

[8] http://www.cs.wm.edu/~denys/data/icpc09

Table 1. Average percentage of the number of methods classified as relevant, somewhat relevant, and not relevant in the top ten results returned by each feature location technique for *jEdit*, *Eclipse*, and both.

	jEdit			*Eclipse*			Both Systems		
Feature location technique	**Relevant**	**Somewhat Relevant**	**Not Relevant**	**Relevant**	**Somewhat Relevant**	**Not Relevant**	**Relevant**	**Somewhat Relevant**	**Not Relevant**
IR_{query} [14]	12.5%	15%	72.5%	22.5%	12.5%	65%	17.5%	13.75%	68.75%
IR_{seed}	12.5%	20%	67.5%	12.5%	22.5%	65%	12.5%	21.25%	66.25%
IR_{query} + Dyn_{marked} [13]	30%	20%	50%	25%	5%	70%	27.5%	12.5%	60%
IR_{query} + Dyn_{full} [15]	15%	22.5%	62.5%	25%	12.5%	67.5%	17.5%	17.5%	65%
IR_{seed} + Dyn_{marked}	20%	15%	65%	27.5%	25%	47.5%	23.75%	20%	56.25%
IR_{seed} + Dyn_{full}	15%	27.5%	57.5%	27.5%	35%	42.5%	18.75%	31.35%	50%
IR_{query} + Dyn_{marked} + Static	30%	17.5%	52.5%	30%	12.5%	57.5%	30%	15%	55%
IR_{query} + Dyn_{full} + Static [6]	12.5%	25%	62.5%	30%	12.5%	57.5%	21.25%	20%	58.75%
IR_{seed} + Dyn_{marked} + Static	17.5%	17.5%	65%	30%	15%	55%	23.75%	25%	51.25%
IR_{seed} + Dyn_{full} + Static	12.5%	30%	57.5%	27.5%	22.5%	50%	20%	26.25%	53.75%

generated better results for *Eclipse*, therefore, it is possible that static analysis was able to explore the PDG more fully and find more relevant methods in *Eclipse* than *jEdit*.

The purpose of this exploratory study was to learn how effective feature location techniques are at finding multiple methods relevant to a feature. If we had set out to find only a single method associated with a feature, the techniques we evaluated performed with effectiveness comparable to that reported in previous studies [13, 15]. On average in *jEdit*, at least one relevant method was found in the top ten for each feature by every technique. In *Eclipse*, all but one approach had at least 20% of its top ten methods categorized as relevant. Most approaches found closer to 30%. These results are more encouraging than those for *jEdit*, but they still allow room for improvement.

4.2. Research question 2

Based on the data from both systems, there is no consensus on whether an *nl-query* or a *method-query* is better. This result suggests that using an automatically generated query of identifiers from a seed method performs just as well as a query constructed by a human, which could eliminate much of the subjectivity inherent in formulating queries for feature location. Even though there is no clear winner, some interesting observations can still be drawn. The *nl-queries* consisted of a few words, while the *method-queries* were comprised of many identifiers. The larger the seed methods, the more identifiers there generally were. In *jEdit*, the seed methods varied in size from 9LOC and fewer than 20 identifiers to 147LOC and over 100 identifiers. The wealth of identifiers in larger methods may aid textual analysis by providing more query terms, but this trend is not universal. The seed for one feature had over 100 terms, but the two types of queries performed the same.

4.3. Research question 3

On average, the use of *marked* traces produced better results than *full* traces, which supports the findings of a previous study [13]. Using *marked* traces limits the number of methods recorded as executed, meaning more methods will be pruned from a ranked list. On the other hand, *full* traces were better at finding methods categorized as somewhat relevant. The methods classified as somewhat relevant generally seem to be in the call chain of relevant methods but do not directly implement a feature. We can offer no explanation for why *full* traces found more somewhat relevant methods and conjecture it may be coincidental.

4.4. Threats to validity

There are several issues that may limit the generalizability of our results. Foremost is the subjectivity in the evaluation. To minimize bias, the author did not know to which approach each top ten list belonged. Also, we formalized how methods were classified by creating guidelines. For one feature, we asked several programmers to categorize methods and compared them to the author's. Since the classification agreement with the author was high, it is reasonable to assume that the author's classifications are sound. Another subjective aspect of this work is the construction of *nl-queries* and the selection of seed methods. To form *nl-queries*, we used words from the change requests and bug reports. The seed methods were randomly selected from patches to the features/bugs. The use of other queries and seeds could have altered the results.

Another threat is that only one scenario was used to collect traces. Every effort was made to ensure that the scenarios fully captured the behavior of the features, but aspects may have been missed. Finally, we only studied a small number of features from two systems, both written in Java, limiting the ability to generalize our results to other types of systems. *Eclipse* is a real-world system, but *jEdit* is rather small in comparison. This threat can be reduced if we experiment on a larger number of diverse systems.

5. Related work

This section reviews some existing feature location approaches by categorizing them as static, dynamic, or hybrid. A more complete discussion of feature location techniques can be found in [1].

Most static feature location techniques are either structural or textual. Structural approaches include [4, 11].

Textual approaches utilize such techniques as information retrieval [14, 16], independent component analysis [9], and natural language [20]. Some tools use both structural and textual information to locate code [10, 22] by using textual information to prune structural relationships, or vice versa.

Some of the earliest work on feature location was software reconnaissance [21], a dynamic approach that compares a trace of a program when a feature is invoked to a trace when it is not. This approach has been expanded and improved [1, 8]. Hybrid feature location leverages the benefits of static and dynamic analyses. Eisenbarth et al. [7] developed a technique that applies formal concept analysis to traces to produce a mapping of features to methods. In PROMESIR [15], LSI is combined with a dynamic technique known as SPR [1] to rank methods likely relevant to a feature. In SITIR [13], a single execution trace is filtered using LSI to extract code relevant to a feature.

Cerberus [6] is the only approach we are aware of that combines three types of analyses for feature location. Cerberus does not produce a ranked list of methods, while all the techniques we studied do. We investigated several combinations of information because Cerberus is not able to locate methods relevant to some features.

6. Conclusion

This paper presented an exploratory study of the effectiveness of ten feature location approaches at finding near-complete implementations of features. Although we did not discover an approach that outperforms all others, we did observe that combining analyses generally improves results. One promising result is that *method-queries* perform comparably to queries formed by a human. We also summarized cases in which certain combinations of analyses were more effective than others. These findings can be used in future research to improve feature location.

Acknowledgements

We acknowledge David Coppit, Andrian Marcus, and Václav Rajlich for contributions to previous versions of this research and Huzefa Kagdi for his helpful comments. We thank Maksym Petrenko for his help with JRipples and the students who took part in the study. This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

References

[1] Antoniol, G. and Guéhéneuc, Y. G., "Feature Identification: An Epidemiological Metaphor", *TSE*, vol. 32, no. 9, 2006, pp. 627-641.

[2] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in Proc. of ICSE'94 May 17-21 1994, pp. 482-498.

[3] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., "JRipples: A Tool for Program Comprehension during Incremental Change", in Proc. of IWPC'05, May 15-16 2005, pp. 149-152.

[4] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of IWPC'00, June 2000, pp. 241-249.

[5] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[6] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in Proc. of ICPC'08, Amsterdam, The Netherlands, 2008.

[7] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *TSE*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[8] Eisenberg, A. D. and De Volder, K., "Dynamic Feature Traces: Finding Features in Unfamiliar Code", in Proc. of ICSM'05, Budapest, Hungary, September 25-30 2005, pp. 337-346.

[9] Grant, S., Cordy, J. R., and Skillicorn, D. B., "Automated Concept Location Using Independent Component Analysis ", in Proc. of WCRE'08, Antwerp, Belgium, 2008.

[10] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of ASE'07, November 2007, pp. 14-23.

[11] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence", in Proc. of ICPC'07, Banff, Canada, June 2007.

[12] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension", *IEEE Softw.*, vol. 3, no. 3, 1986, pp. 41-49.

[13] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of ASE'07, November 5-9 2007.

[14] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of WCRE'04, Delft, The Netherlands, Nov. 9-12 2004, pp. 214-223.

[15] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *TSE,* vol. 33, no. 6, June 2007, pp. 420-432.

[16] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of ICPC'07, Banff, Alberta, Canada, June 2007.

[17] Robillard, M. P., "Topology Analysis of Software Dependencies", *TOSEM*, vol. 17, no. 4, August 2008.

[18] Robillard, M. P., Shepherd, D., Hill, E., Vijay-Shanker, K., and Pollock, L., "An Empirical Study of the Concept Assignment Problem", McGill University June 2007.

[19] Salah, M. and Mancoridis, S., "A hierarchy of dynamic software views: from object-interactions to feature-interactions", in Proc. of ICSM'04, Chicago, IL, September 11-14 2004, pp. 72-81.

[20] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proc. of AOSD'07, 2007, pp. 212-224.

[21] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.

[22] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *TOSEM*, vol. 15, no. 2, 2006, pp. 195-226.

[23] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *TSE*, vol. 31, no. 6, June 2005, pp. 429-445.