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 insight into situations when certain feature location approaches work well and then they fall short. Our results and observations can be used to guide future research on feature location techniques that will be able to find near-complete implementations of features.

1. Introduction

Software maintenance and evolution tasks first require programmers to understand the implementation of specific parts of an existing software system [18]. To do so requires locating the source code that implements functionality, an activity known as *concept assignment* [2] or *feature location* [31]. Most existing feature location techniques are quite effective at finding a starting point of a feature's implementation, i.e., one method that is relevant to that feature [16, 19, 20]. However, it is rarely the case that a single method is the sole contributor to a feature. These techniques leave it up to programmers to find the other methods that implement 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 denote 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. One programmer may consider a method relevant, while another may not [23].

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 the *jEdit* and *Eclipse* software systems. As part of the assessment, we designed easy-to-follow guidelines for evaluating feature location techniques. Additionally, we explored a new mechanism for formulating queries used by textual analysis that automatically constructs a query from the identifiers of a method. All our data is made publicly available on a web site.

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 means of automatically creating a query for textual analysis performs comparably to a query formed by a human. Overall, the results of this exploratory study can be used to improve the development of feature location techniques so that they are able to find near-complete implementations of features.

2. Feature location techniques

A *feature* is a functional requirement of a program 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, 9].

Feature location is the activity of identifying the source code elements (i.e., methods) that implement a feature [31]. We investigate several approaches to locate the source code associated with a feature using textual, dynamic, and static analyses. Next, we explain each type of analysis and how we combined them in this work.

2.1. Core techniques

Textual analysis. The implementation of a feature, even if dispersed among many methods, may use a consistent vocabulary in terms of identifiers and the words appearing in comments. One approach to locate features is to determine textual similarities among a user query and source code elements (e.g., methods). A query is a set of words formulated by a user to describe a feature. Alternatively, a query can be automatically comprised of the identifiers and comments in a method that is known to be relevant to a feature. In either case, textual analysis and feature location can be performed using the information retrieval technique known as Latent Semantic Indexing (LSI) [7]. With LSI, the relation between terms (words) and documents (methods) can be discovered. In brief, comments and identifiers are extracted to form a corpus. LSI indexes the corpus and creates a signature for each document (method), and these indices are used to define similarity measures between methods. Users can formulate queries in natural language (*nl-queries*) or by using the identifiers of a known relevant method (*method-queries*). LSI returns a list of all the methods in the software ranked by textual similarity to the query. An advantage of this approach is that a working version of the source code is not required. However, if a program's identifiers are not meaningful, the ranking results can be affected.

For large systems, a ranked list with thousands of methods is a formidable amount of information, unless the majority of the methods that implement the feature appear near the top of the ranked list. Often, a threshold is set to limit the number of methods that users consider. The threshold may be set by a cut point, as in only the top *n* or only the top *x* percent of results are considered. The threshold can be set as at a specific value such that only the results with a similarity greater than or equal to the threshold are considered. Determining an appropriate threshold is an open research problem.

Dynamic Analysis. Using dynamic information is another approach to feature location [31, 32]. Dynamic information complements textual information since not all methods relevant to a feature may use a similar vocabulary, but they may be executed when a system is run. To collect dynamic information, an executable version of the system must be available. Users develop *scenarios* that trigger a feature. A scenario is a sequence of user inputs to a system. As scenarios are being exercised, *traces* can be collected. A trace is a list of events that occurred within the system's execution. Events can be method invocations, object instantiations, and variable accesses. In this work, we focus on method calls only.

There are two types of traces that we consider. A *full trace* [31] captures all events from a system's start-up to shutdown. A *marked trace* [16, 25] only captures events during part of a system's execution. When the system is running, users can start and stop tracing. By starting tracing immediately before triggering a feature and stopping tracing once the feature's behavior is observed, more of the events (methods) listed in the trace should pertain to the feature.

Static Analysis. Dynamic information is only as good as the scenarios used to collect traces. If scenarios fail to invoke a feature in a certain way, relevant methods may be missing from an execution trace. Since static analysis does not rely on a program's execution, statically collected information can compensate for dynamic analysis' weaknesses [11]. Static analysis can provide a wealth of information on different types of dependencies such as control flow, data dependence, and inheritance. For this work, we use light-weight static analysis and focus on method caller-callee relationships by using a static **p**rogram **d**ependency **g**raph (PDG) [5, 13, 21] in which nodes are methods and edges represent method invocations. We obtain such a graph using JRipples[1][4].

Additional methods relevant to a feature can be found by exploring a PDG. Starting at a *seed method*, one that is known to be relevant to a feature, other methods pertinent to the implementation of the feature can be discovered by traversing the graph. Executing a program may not invoke a relevant method, but if that relevant method has a static dependency with the seed method, static analysis can locate it. However, in the case that a method related to a feature has no static dependencies with the seed, static analysis fails.

2.2. Combined techniques

Textual, dynamic, and static information compliment each other, so in theory when working in tandem, they should produce better results than when used individually. In this work, we investigate the following combinations of analyses.

Textual Analysis. The first feature location technique we consider employs only textual analysis, and we consider it to be our baseline approach. We evaluate two configurations of textual analysis, one using *nl-queries* as in [16] and one 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 to feature location was introduced in [17], whereas IR_{seed} is a new version of this technique.

Textual Analysis plus Dynamic Analysis. We also examine the combination of textual and dynamic analysis for feature location. To combine these two types of information, methods that are not executed are removed from the ranked list provided by textual analysis as in [16]. We investigate all configurations of the different types of queries and traces. Abbreviated, these configurations are $IR_{query} + Dyn_{marked}$, $IR_{query} + Dyn_{full}$, $IR_{seed} + Dyn_{full}$, and $IR_{seed} + Dyn_{full}$, where "Dyn" stands for dynamic analysis and subscripts denote the type of trace (i.e., full and marked). $IR_{query} + Dyn_{marked}$ is like the

---

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

approach presented in [16], while IR_{query} + Dyn_{full} is similar to [19]. The two other approaches are novel combinations.

Textual, Dynamic, and Static Analyses. The final feature location technique we evaluate incorporates all three types of analyses. Again, we investigate all configurations of queries and traces in conjunction with static analysis: 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 [8], but instead of using prune-dependency analysis, it uses light-weight static analysis. The other three combinations are new.

Unlike with combining textual and dynamic analysis, utilizing static analysis does not involve pruning an existing ranked list. Instead, static analysis entails exploring a PDG to find relevant methods and then ranking them once exploration stops. Searching begins at a seed method known to be relevant to the feature. The static neighbors of the seed (parents and children) are examined to see if they meet the textual and dynamic criteria. The textual criterion is a threshold similarity value, and the dynamic criterion is whether the method appears in a given trace. If the method's textual similarity is above the threshold and it was executed, it is added to the list of results, and its neighbors are added to a list of methods to be examined. Once the list of methods to examine is empty, exploration stops and the list of results is sorted by textual similarity values.

Cerberus [8] uses all three types of analyses. We did not use Cerberus because it does not produce a ranked list of methods and the other techniques in our evaluation do. Therefore for the sake of comparison, we developed our own combination of textual, dynamic, and static analyses.

In total, we investigate ten different feature location techniques, many of which are novel because they involve *method-queries*. There are other possible combinations of textual, dynamic, and static analysis that we decided not to study, such as just dynamic and static analysis together. We decided against including these other approaches in our study since they do not produce a ranked list and the results of using standalone versions of static and dynamic analyses are available elsewhere [5, 16, 19]. The details of how we evaluated and compared the ten approaches described above are provided in the next section.

3. Exploratory study

We performed an exploratory study to evaluate the feature location techniques mentioned above. The goal of the study was to determine which combination of analyses provides the best results and under what circumstances. This section outlines the subject systems used in the study, our research goals, and the specifics on how we used each type of analysis.

3.1. Research questions

We set out to seek the answers to a number of research questions in this exploratory study. These **r**esearch **q**uestions (RQ) are as follows:

- **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*, is better at discovering numerous methods that implement a feature?

Since this study was exploratory in nature, we did not know what to expect as the outcome, so we did not formulate any hypotheses. However, intuition and previous research results led us to conjecture that the approaches that incorporated more types of analyses would perform better than those with fewer.

3.2. Subject software systems

For our study, we chose two open-source Java software systems of different sizes and from different domains. *jEdit*[2] is a highly configurable and customizable text editor. We used version 4.3pre16 in our study, which consists of approximately 105KLOC in 910 classes and 5,530 methods. We selected four features from *jEdit* to study. These features were chosen from feature requests with submitted patches in the "Patches" section of the systems' tracking software.

- **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).

---

[2] http://www.jedit.org/ (verified on 09/18/08)

Table 1. Queries, scenarios and seed methods used for each feature in the case study.

Feature	Description	Query	Scenario	Seed Method
jEdit Patch #1608486	Support for "thick" caret.	configuration global option thick caret text area block	Start jEdit; click "Global Options" button then "Text Area;" start tracing; click "thick" checkbox then "OK;" stop tracing; exit.	EditPane.initPainter (49LOC, 114 terms)
jEdit Patch #1818140	Edit the entries in the History Text.	history text edit string menu	Start jEdit; click "Find" button; start tracing; right click in text area; select "Previously entered searches;" delete, insert, and modify an entry; click "OK;" stop tracing; click "Close;" exit.	ListModelEditor.createTableModel (9LOC, 18 terms)
jEdit Patch #1923613	Reverse searching with regular expressions.	reverse regex search regular expression	Start jEdit; place cursor at end of file; start tracing; click "Find" button; select "Regular Expressions" and "Backwards;" enter "[0-9]+;" click "Find" several times; stop tracing; exit.	SearchDialog.updateEnabled (30LOC, 53 terms)
jEdit Patch #1849215	Match angle brackets.	angle right find next	Start jEdit; place start tracing; cursor to right of "<" whose match is on same line; place cursor to right of "<" whose match is on another line; stop tracing; exit.	TextUtilities.findMatchingBracket (147LOC, 117 terms)
Eclipse Bug #5138	Double-click-drag to select multiple words.	mouse double click up down drag release	Start Eclipse; start tracing; click and release the mouse button; click a second time quickly and hold the mouse button down, drag and select some text; release the mouse button; stop tracing, exit.	TextViewer.mouseUp (11LOC, 36 terms)
Eclipse Bug #31779	UnifiedTree should ensure file/folder exists.	unified tree node file system folder location	Start Eclipse; start tracing; create a file from the file system in a project; refresh; stop tracing; exit.	UnifiedTree.addChildren (53 LOC, 108 terms)
Eclipse Bug #19819	Emacs-style incremental search.	incremental search	Start Eclipse; start tracing; press Ctrl+J; type search criteria; use up and down arrow keys to find matches; stop tracing; exit.	IncrementalFindAction.run (14LOC, 23 terms)
Eclipse Bug #32712	Repeated error message.	delete resource project file folder fail	Start Eclipse; create a simple project; add a file; edit foo.doc externally; start tracing; delete the project; stop tracing, exit.	ResourceTree.standardDeleteProject (78LOC, 216 terms)

Eclipse[3], the other system in our study, is a popular integrated development environment. We used version 2.1, and it contains approximately 2.3MLOC in over 7,000 classes and 89,000 methods. Like with *jEdit*, we selected four features from its bug tracking system. With *Eclipse*, we chose fixed bugs corresponding to misbehaving features. These bugs are:

- **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).

3.3. Input to the analyses

Textual Analysis. We formulated the *nl-queries* used by textual analysis by reviewing the description and comments in the thread for the patch/bug in *jEdit* and *Eclipse*'s tracking systems. The *nl-queries* are listed in Table 1. The *method-queries* consist of the identifiers from seed methods also listed in the table. The seed methods were randomly chosen from the patch for each feature to ensure that they do actually pertain to the feature.

Dynamic Analysis. We created one usage scenario per feature to collect traces in this study. Descriptions of the scenarios are in Table 1. We devised the *jEdit* scenarios by reading the description and comments for the patch in the bug tracking software. For *Eclipse*, two bug reports (#5138 and #32712) had steps to reproduce the errors, and those steps were used as the scenarios for those two features. The scenario for bug #31779 is reused from [16]. For Bug #19819, a scenario was created in which the behaviors of the incremental search feature, as described in the bug report, were exercised.

Static Analysis. The seed methods used as the starting point of static analysis are listed in Table 1. They were the same methods used for constructing the *method-queries* and were randomly selected from the feature's patch. As explained in Section 2.2, static analysis starts at the seed method and branches out in part based on a textual similarity threshold. To determine the textual similarity threshold to set for examining neighbors in a PDG, we adapted the gap threshold technique [17, 33]. A gap threshold is found by determining the largest difference between two adjacent textual similarity values in a ranked list. The threshold is set as the larger of the two values at this location in the list. We adapted this technique to incorporate a relaxation strategy. If the size of a ranked list did not reach our minimum (e.g., ten items), then we decreased the threshold by 0.05 and repeated the procedure again.

---

[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

3.4. Relevancy Assessment

Each combination of analyses is a feature location technique that produces a ranked list of methods suggested to be relevant to a feature. We restrict our evaluation to the top ten methods on each list because other researchers have shown that users are generally unlikely to look at more than ten elements on a list [18, 34]. If most of the top ten suggestions provided by a feature location approach are false positives, then the effort that would be needed to examine more results lower in the list is likely to not be worth the cost.

In reviewing the top ten methods returned by each technique, there needs to be well-defined criteria for judging whether a method is relevant to a feature or not. In almost all cases, the methods that implement a feature are not documented; otherwise feature location would not be necessary. Therefore, other ways of determining a method's relevance to a feature are needed. One option is to present the top ten suggestions to an expert. If no expert is available, then if a bug related to the feature has been fixed, the methods in the patch can be used. However, the bug may only pertain to a small subset of the feature's relevant methods, so relying on a patch may give an incomplete picture of a feature's implementation. For this reason, we decided not to use this evaluation approach, even though we had patches for each feature.

An alternative is to ask programmers to identify relevant methods by exploring the source code. Robillard et al. [23] provided some guidance to participants asked to locate methods relevant to features. The participants were instructed to decide if a method was relevant by asking if it would be useful to know if the method was related to the feature if the feature had to be modified in the future. We take a similar but adapted approach in our evaluation. Instead of asking programmers to locate relevant methods on their own, we present them with lists of methods and ask them to determine the relevance of each method. In our study, the participants were provided with code and an executable, a description of a feature and how to invoke it, and the following guidelines for how to determine if a method is relevant to the feature or not.

1. Method names that are similar to the words in the feature's description are good indicators of possibly 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, i.e. what other methods call it and are called by it. If the method's dependencies seem relevant, then the method probably is also.

Having a number of programmers follow these guidelines and focusing on the agreement between the programmers eliminates any one individual's bias.

One of the authors classified every method in the resulting ranked lists for all eight features without knowing which technique produced each list. To give support to the resulting categorizations, we solicited volunteers to also classify methods and compared them to the author's. Four students volunteered to participate in this study. The students were enrolled in a graduate software engineering course. They were given ten ranked lists each containing ten methods. The ten lists corresponded to the ten different feature location techniques under evaluation. The students were not aware to which feature location technique the lists pertained. They were instructed[8] to classify the methods based on the guidelines above, and *jEdit*'s *thick caret* feature was used. The patch for this feature has six methods, and the feature location techniques were able to find between one and three of these methods in the top ten of their ranked lists.

Figure 1 shows the average agreement between the author's classifications and the volunteers. To demonstrate how percent agreement was calculated, consider the following example. The author classified four methods from a list of ten as relevant and six as not relevant, and a volunteer classified only three methods as relevant and seven as not relevant. The volunteer's three methods were included in the four identified by the author, so the percent agreement is 90%. Nine out of ten times, both programmers agree that a method either belonged in the relevant or not relevant categories. The percent agreement was averaged over all ten lists generated by the different feature location techniques. When computing agreement between more than two programmers, all individuals involved had to categorize a method in the same way for there to be agreement. The percent agreement between the author and the volunteers is high; it is always greater than 70%. The agreement declines only slightly when more individuals are taken into account. Agreement about relevant methods was highest, followed by agreement about irrelevant methods, suggesting that it is easiest to identify methods that definitely do or do not implement a feature.

---

[8] See http://www.cs.wm.edu/~meghan/case-study-instructions.html

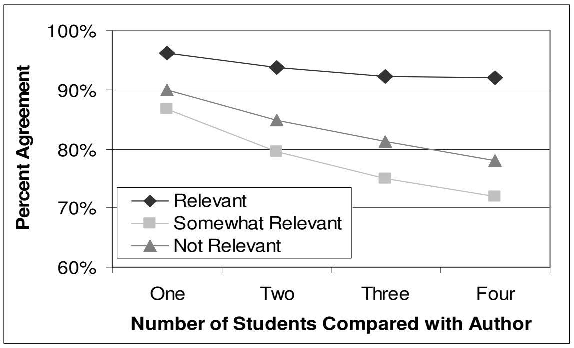

Figure 1. Percent agreement of the volunteers with one of the authors for the *jEdit thick caret* feature.

The average agreement among programmers about a method's relevance in this study was higher than that observed by Robillard et al. in [23]. The two approaches to evaluating method relevance differ: our study provided lists of methods for programmers to judge while Robillard et al. asked programmers to find the methods implementing a feature themselves. Also, our study allowed programmers to place methods into one of three categories to allow for uncertainty instead of a binary yes/no classification.

4. Results

In our study, only the top ten ranked methods returned by a feature location technique for each feature were examined. Those methods were then classified into three categories (relevant, somewhat relevant, not relevant) as described in the previous section. The results of the *jEdit* and *Eclipse* studies are discussed in the next sections and are also available online[9].

4.1. jEdit study findings

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 2. An in-depth discussion of the results is below.

RQ1. For *jEdit*, the techniques that found the most relevant methods on average were IR_{query} + $\text{Dyn}_{\text{marked}}$ and IR_{query} + $\text{Dyn}_{\text{marked}}$ + Static with 30% of the top ten methods being relevant, meaning three methods in the top ten were relevant on average. These approaches found nearly double the amount of relevant code than most of the other techniques which averaged between 12.5% and 20%. Different programmers may consider the methods classified as somewhat relevant as pertaining to the implementation of a feature, while others might not. If the somewhat relevant methods are considered important to a feature's implementation, then IR_{query} + $\text{Dyn}_{\text{marked}}$ is the best performing technique in the *jEdit* study with 50% of the located methods being relevant on average. At least for *jEdit*, the IR_{query} + $\text{Dyn}_{\text{marked}}$ feature location technique is readily able to locate many methods implementing a feature and not just a single method.

Since the IR_{query} + $\text{Dyn}_{\text{marked}}$ and IR_{query} + $\text{Dyn}_{\text{marked}}$ + Static approaches performed the same, these results suggest that adding static analysis provides no additional benefits over a combination of only textual and dynamic analysis. However, the approach that located the most relevant methods for the *edit history text* feature was IR_{query} + $\text{Dyn}_{\text{marked}}$ + Static. Seventy percent of the methods in its top ten list were relevant. The methods implementing this feature have very clear structural dependencies because they can be found along the same branch of the PDG. Therefore, static analysis was easily able to identify multiple methods related to this feature. With the three other *jEdit* features, static analysis did not perform as expected and improve the number of relevant methods located. Incorporating static analysis yielded no more relevant methods than using a combination of textual and dynamic analysis. For *jEdit*'s *reverse regex search* feature, a different seed than the one listed in Table 1 was originally selected. However, the seed method was isolated in the PDG, so static analysis could not expand far beyond it to locate more potentially relevant methods. This is one of the observed limitations of static analysis for feature location.

Another reason static analysis may not produce improved results is even when there is a dependency between a seed method and a relevant method, they may be distant from each other in the PDG. If one method along a branch in a PDG between the seed and a relevant method is not executed or has a textual similarity below the threshold, static analysis will be unable to locate the relevant method. Therefore, the ranked list is populated with other, irrelevant methods that meet both the textual and dynamic criteria when searching the PDG.

In general, combining just textual and dynamic analysis either did not affect the number of relevant methods located (*reverse regexp* feature) or slightly improved the results (*edit history text* and *angle bracket matching* features) by pruning unexecuted methods from the ranked list. This result supports the findings of previous studies [16]. However, the combination of the two analyses did not find a substantial number of relevant methods for each feature.

For *jEdit*'s *thick caret* feature, surprisingly, we observed that adding dynamic analysis to textual produced worse results than textual analysis alone. The StandaloneTextArea.initPainter method appears to be a code clone of EditPane.initPainter, the seed method, meant to be used when *jEdit* is embedded in another system. The IR_{seed} approach locates this method, but

---

[9] http://www.cs.wm.edu/~meghan/case-study-results.html

Table 2. 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} [17]	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} [16]	30%	20%	50%	25%	5%	70%	27.5%	12.5%	60%
IR_{query} + Dyn_{full} [19]	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 [8]	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%

neither the IR_{seed} + Dyn_{marked} nor the IR_{seed} + Dyn_{full} approach can identify this method because it was not executed. This case highlights a challenge associated with using dynamic analysis for feature location. One solution is to create a better scenario, or perhaps when combining textual and dynamic analysis, if a method has a high enough textual similarity, the fact that it was not executed should be ignored.

Our goal was to locate as many methods as relevant to a feature as possible. If we had set out to find only a single method to use as a starting point for searching for more methods associated with a feature, the techniques we evaluated performed with effectiveness comparable to that reported in previous studies [16, 19]. On average, at least one relevant method was found in the top ten for each feature by every technique. However, since the average number of relevant methods found by the feature location techniques is low, this work highlights the fact that finding a near-complete set of methods that implement a feature is not simple.

RQ2. Based on the *jEdit* data, there is no consensus on whether an *nl-query* or a *method-query* is best. For the *reverse regexp* feature, the *nl-query* performed better, while for the *thick caret* feature, the *method-query* was best. For the two other features, both queries returned the same number of relevant methods. 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 a query.

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. The seed methods (refer to Table 1) varied in size from 9LOC and fewer than 20 identifiers (*edit history text*) to 147LOC and over 100 identifiers (*match angle brackets*). Considering only the IR_{query} and IR_{seed} results, the *method-query* for the *thick caret* feature (114 terms) performed better than the *nl-query* (8 terms) with 30% relevant vs. 10%. 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 the *angle bracket matching* feature has over 100 terms, but the two types of queries performed the same.

RQ3. On average, the use of *marked* traces produced better results than *full* traces when locating relevant methods for features in *jEdit*, which supports previous studies as well [16]. Using *marked* traces limits the number of methods that appear to be executed, meaning more irrelevant 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 the feature. We can find no explanation for why *full* traces found more somewhat relevant methods and conjecture it may be coincidental.

The nature of a feature should be considered before deciding to use *marked* traces over *full* traces. A feature like *angle bracket matching* that does not have a menu interface is suitable for *marked* traces, but for features that involve setting options in a dialog or menu, like *jEdit*'s *thick caret* and *reverse regex* features, *full* traces might be the better option. Consider the method TextAreaOptionPane._init that adds various options for *jEdit*'s main text area, including the thick caret option, to a dialog. This method was executed, but it did not appear in the *marked* trace since tracing was started after the dialog opened. *Marked* traces run the risk of omitting initialization code that *full* traces would not.

4.2. Eclipse study findings

Table 2 lists the average number of relevant, somewhat relevant, and not relevant methods found in the top ten lists of each technique in *Eclipse*. Below, we discuss the results with regards to our research questions.

RQ1. For *Eclipse*, there were three approaches that, on average, performed the best at finding relevant methods: IR_{query} + Dyn_{marked} + Static, IR_{query} + Dyn_{full} + Static, and IR_{seed} + Dyn_{marked} + Static. Thirty percent of the top ten methods identified were relevant. When taking both relevant and somewhat relevant methods into account, the best performing approach was IR_{seed} + Dyn_{marked} + Static, with on average 62.5% or slightly better than six methods out of the top ten.

Unlike in *jEdit*, these results suggest that static analysis does aid feature location. Examining individual features, a mixed story emerges. For bugs #19819 and #32712, adding static analysis produced no improvement over a combination of textual and dynamic analysis. Bug #5138 actually saw the number of relevant methods decrease when static analysis was used. Combining textual and dynamic analysis essentially 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 one that uses static analysis. This is exactly what happened in the case of bug #5138. The seed method was isolated in the PDG, so static analysis was not able to branch out very far.

Feature location on bug #31779 resulted in the biggest improvement when adding static analysis. Ninety percent of the methods on the top ten list for IR_{query} + Dyn_{marked} + Static were relevant, while 100% of the methods for IR_{query} + Dyn_{full} + Static were. Static analysis was able to succeed so well with this feature because many of the relevant methods were located in the same class as the seed. The results for these two approaches for this feature may have skewed *Eclipse*'s averages. Nevertheless, this case is an example that it is possible to locate near-complete feature implementations and that static analysis is a useful tool to do so.

Overall, the combination of textual and dynamic information improved results over only textual analysis, but for one feature the use of textual and dynamic information caused the number of relevant methods located to decrease. The IR_{query} technique identified JavadocDoubleClickStrategy.doubleClicked as relevant to bug #5138. However, this method is not executed in the scenario because no Javadoc comments were double clicked. Therefore, this method has no chance of being identified by an approach that uses dynamic analysis unless a new scenario is used. Alternatively, since this method has a high textual similarity, a revised combination of textual and dynamic analysis that allows for cases when a method is unexecuted but has high similarity could also solve this problem.

The purpose of this exploratory study was to learn how effective feature location techniques are at finding multiple methods relevant to a feature instead of just a single starting point. In *Eclipse*, all but one approach had at least 20% of its top ten located methods categorized as relevant. Most approaches found closer to 30%. These results are more encouraging and those for *jEdit*, but they still show room for improvement. Being able to fully locate the implementation of a feature is a difficult problem that requires further research.

RQ2. The data showed that *method-queries* perform comparably to *nl-queries.* This outcome is similar to what was observed in *jEdit*. Considering the IR_{query} and IR_{seed} results for bugs #5138 and #19819, the seed methods were short (11LOC/36 terms and 14LOC/23 terms), and *nl-queries* performed better for these features. For bug #31779, the two types of queries achieved comparable results to each other, but for bug #32712 (78LOC/216 terms), the *method-query* was the winner. This possible trend of *method-queries* from longer methods performing better was also seen in *jEdit*, adding weight to the idea of automatically constructing queries from the identifiers of seed methods.

RQ3. In *Eclipse*, *marked* traces outperformed *full* traces slightly. When the same type of query was used, *marked* traces found about 5% more relevant methods than *full* traces. We attribute this outcome to *marked* traces limiting the method invocations recorded, thus removing much noise from the resulting trace. Collecting *full* traces is difficult because they are very larger and take time to collect, especially for a system like *Eclipse*. This fact plus the better performance of *marked* traces make them the ideal choice in most cases.

4.3. Discussion

Based on this exploratory study, we draw a number of notable conclusions, which are discussed below.

***Method-queries* perform as well as *nl-queries*.** There was no clear winner when it comes to *nl-queries* vs. *method-queries*. This result is promising because it means that automatically generated queries perform just as well as ones created by humans. We observed that *method-queries* from larger seeds seem to perform the best. These results motivate further exploration into strategies for formulating queries automatically.

No feature location technique is universally successful at finding near-complete implementations of features. At best, they are good at locating a few relevant methods. This research motivates the need for feature location techniques that successfully discover as many feature-relevant methods as possible.

The effectiveness of static analysis might be tied to the effectiveness of textual analysis. The biggest difference between the results of the two systems concerns the use of static analysis. In *jEdit*, feature location with static analysis did not produce better results than approaches without it. In *Eclipse*, the best techniques used static analysis. One possible reason for this discrepancy stems from textual analysis. Static exploration of a PDG was performed using textual and dynamic criteria. If a method did not meet the textual similarity threshold, then exploration down that path of the PDG would halt. LSI generated better results for *Eclipse*, therefore, it is possible that static analysis was able to explore a PDG more fully and find more relevant methods in *Eclipse* than *jEdit*. Using additional types of static dependencies along with light-weight analysis may improve results.

Marked **traces slightly outperform** ***full*** **traces.** In both systems, *marked* traces were able to find slightly more relevant methods than *full* traces due to the fact that *marked* traces capture a higher concentration of feature-relevant methods. However, *full* traces should be used for features that are invoked through menus.

LSI performs better on larger systems. One difference between the results of the two systems is that textual analysis yielded better results in *Eclipse*. There are two possible reasons for this outcome. First, *Eclipse* is a professional-grade system, so the naming conventions used may be stricter than in *jEdit*, which would aid LSI. Another possible reason is that the performance of LSI has been shown to degrade on smaller corpora [8]. *jEdit*'s corpus is small (about 7K terms and 5K methods) in comparison to *Eclipse*'s (56K terms and 89K methods), therefore LSI's ranking strategies may be more effective with *Eclipse*.

The textual similarity threshold selected by gap technique was too high. We adapted the gap threshold technique with a relaxation strategy in the case fewer than ten methods were found. The initial textual similarity selected was always too high. The relaxation strategy that we incorporated had to be used in every feature location technique involving static exploration of a PDG. In each case, the threshold had to be lowered significantly, sometimes by as much as 0.5. This observation suggests that feature-relevant methods are not always located close to each other in a PDG.

4.4. Threats to validity

There are several issues that may limit the generalizations that can be drawn from our results. Foremost is the subjective manner in which the results were judged. One author determined the relevance of the methods found by the feature location techniques. 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 also asked several programmers to categorize the methods and compared them to the author's. Since the agreement between the author and the students was high, it is reasonable to assume that the author's classifications are representative of the features.

Another subjective aspect of this work is the construction of the *nl-queries* and the selection of the seed methods. To form the *nl-queries*, we used words from the change requests and bug reports. The seed methods were randomly selected from methods that were submitted in patches to the features/bugs. Since those methods had to be changed to perform maintenance on the features, they must be relevant to the feature. However, the use of different queries and different seeds could alter the results.

Another threat to validity is that only one scenario was used to collect execution traces. Every effort was made to ensure that the scenarios dependably captured the behavior of the features, although certain aspects may have been missed. In many cases, the scenarios were based on the descriptions given in a bug report.

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 software systems. *Eclipse* is a real-world system, but *jEdit* is rather small in comparison. This threat can be reduced if we experiment on more systems written in other languages and taken from other domains.

5. Related work

Since feature location is an important part of software maintenance, there are many existing techniques. This section reviews some of these approaches by categorizing them as either static, dynamic, or hybrid feature location. A more complete discussion of feature location approaches can be found in [1], while [6, 30] provide comparisons and evaluations of techniques.

Most static feature location techniques are either structural or textual. Structural approaches [3, 5, 15, 22, 29]. Textual approaches use comments and identifiers to locate code relevant to a feature by utilizing such techniques as information retrieval [17, 20], independent component analysis [12], and natural language [27]. A number of tools use both structural and textual information to locate pertinent code [13, 33] by using textual information to prune irrelevant structural relationships, or vice versa.

Some of the earliest work on feature location was software reconnaissance [31], a dynamic approach that compares a trace of a program when a feature is invoked to a trace when the feature is not executed. Software reconnaissance has been recently expanded and improved [1, 10]. Other dynamic approaches focus on feature interactions [26] and aspect mining [28].

Hybrid feature location approaches seek to leverage the benefits provided by both static and dynamic analysis [24]. Eisenbarth et al. [9, 14] developed a technique that is mostly dynamic and applies formal concept analysis to traces to produce a mapping of features to the program's methods. Several approaches combine LSI and dynamic information. In PROMESIR [19], LSI is combined with a dynamic analysis technique known as SPR [1] to give a ranking of methods likely relevant to a feature. In SITIR [16], a single execution trace can be filtered using LSI to extract code relevant to the feature of interest.

Cerberus [8] is the only approach we are aware of that combines three types of analyses for feature location. Our work is different from Cerberus as we are investigating several alternative combinations because Cerberus is not always able to locate methods relevant to some features. We also distinguish ourselves from Cerberus by examining the trade-offs of using textual, dynamic, and static analyzes for feature location and by evaluating our approaches on small and large systems.

6. Conclusion

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

Acknowledgements

We acknowledge David Coppit for contributions to a previous version of this research and Huzefa Kagdi for his helpful comments. We thank Maksym Petrenko for his help with JRipples and the students in the Software Evolution and Maintenance course at W&M for taking part in the study.

References

[1] Antoniol, G. and Guéhéneuc, Y. G., "Feature Identification: An Epidemiological Metaphor", *IEEE 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 IEEE/ACM ICSE May 17-21 1994, pp. 482-498.

[3] Binkley, D., Gold, G., Harman, M., Li, Z., and Mahdavi, K., "An empirical study of the relationship between the concepts expressed in source code and dependence", *The Journal of Systems and Software*, vol. 81, 2008, pp. 2287–2298.

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

[5] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of IEEE IWPC, Limerick, Ireland, June 2000, pp. 241-249.

[6] Cleary, B., Exton, C., Buckley, J., and English, M., "An empirical analysis of information retrieval based concept location techniques in software comprehension", *Empirical Software Engineering* vol. 14, no. 1, 2009, pp. 93-130.

[7] 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.

[8] 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 IEEE ICPC, 2008.

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

[10] Eisenberg, A. D. and De Volder, K., "Dynamic Feature Traces: Finding Features in Unfamiliar Code", in Proc. of 21st IEEE ICSM, Sept. 25-30 2005, pp. 337-346.

[11] Ernst, M., "Static and Dynamic Analysis: Synergy and Duality", in Proc. of ICSE WODA'03, Portland, OR, May 2003, pp. 24-27.

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

[13] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of IEEE/ACM ASE, Nov. 2007.

[14] Koschke, R. and Quante, J., "On dynamic feature location", in Proc. of IEEE/ACM ASE, Long Beach, CA, USA, 2005, pp. 86-95.

[15] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence", in Proc. of IEEE ICPC, Banff, Canada, June 2007.

[16] 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 IEEE/ACM ASE'07, Atlanta, Georgia, November 5-9 2007, pp. 234-243.

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

[18] Petrenko, M., Rajlich, V., and Vanciu, R., "Partial Domain Comprehension in Software Evolution and Maintenance", in Proc. of ICPC, 2008.

[19] 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", *IEEE TSE* vol. 33, no. 6, June 2007.

[20] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of IEEE ICPC'07, Banff, Alberta, Canada, June 2007, pp. 37-48.

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

[22] Robillard, M. P. and Murphy, G. C., "Concern Graphs: Finding and describing concerns using structural program dependencies", in Proc. of ICSE, 2002.

[23] 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.

[24] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis", in Proc. of IEEE ICPC, 2008, pp. 236-241.

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

[26] Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M., "Scenario-driven dynamic analysis for comprehending large software systems", in Proc. of IEEE CSMR'06, March 22-24 2006, pp. 71-80.

[27] 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, 2007, pp. 212-224.

[28] Tonella, P. and Ceccato, M., "Aspect Mining through the Formal Concept Analysis of Execution Traces", in Proc. of IEEE WCRE'04, 2004, pp. 112 - 121

[29] Weigand-Warr, F. and Robillard, M. P., "Suade: Topology-Based Searches for Software Investigation", in Proc. of ICSE, May 2008, pp. 780-783.

[30] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, vol. 65, no. 2, February 15 2003, pp. 105-114.

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

[32] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S., "Locating program features using execution slices", in Proc. of IEEE ASSET, March 1999.

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

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