Supporting Feature-Level Software Maintenance

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

Abstract

The proposed research defines data fusion approaches to support software maintenance tasks at the feature level. Static, dynamic, and textual sources of information are combined to locate the implementation of features in source code. Structural and textual source code information is used to define feature coupling metrics to aid feature-level impact analysis. This paper provides details on the proposed approaches and evaluation strategies as well as some preliminary results.

1. Problem description

Software maintenance and evolution involves making changes to existing source code. Often, these changes involve making modifications to features in order to address maintenance tasks such as fixing bugs or enhancing features. There are many difficulties associated with software maintenance, but two main challenges are the focus of our proposed research. First, before a modification can be made, the code that implements a feature must be located. Second, after a change has been implemented, impact analysis should be performed to determine if other features could be affected by the change.

Identifying the source code that implements a feature, a process known as *concept* [3] or *feature location* [1], is necessary for program comprehension and maintenance. Existing feature location techniques are good at finding a starting point for a feature, that is, a single relevant method [16-18]. However, it is rarely the case that only one method corresponds to a feature, and the task of locating the rest of the feature's methods is left to the developer. Feature location approaches would be more useful if they could find as many methods associated with a feature as possible.

Features do not exist in isolation [15]. Modifying one feature does not guarantee only that feature is affected by the change. In fact, changes made to one feature may have an impact on seemingly unrelated features. To be aware of the consequences a maintenance task might have on other features, impact analysis can be performed by determining the coupling between features. However, the majority of existing coupling metrics are defined for classes, not features. Generally, features are not implemented by a single class and are scattered throughout the methods of a system [10]. Therefore, metrics that focus specifically on capturing the coupling between features are needed to perform feature-level impact analysis.

2. Research goals

Feature location and software coupling are well established research ideas, but both have room for improvement. In our feature location research, we are interested in answering two questions. First, *what sources of information should we combine to most effectively perform feature location*? Second, *what is the best way to combine those sources of information*? Our goal is to develop a data fusion approach to feature location that incorporates multiple sources of information (such as structural, textual, or dynamic) and produces the most useful results. In our feature coupling work, our goal is to fill a gap in the research area of coupling measurement. There are many options for class-level coupling metrics, but few for feature-level coupling measures. Our goal is to develop metrics that specifically capture feature coupling.

3. Related work

There are many existing feature location techniques that can be broadly classified by the types of information they use: static, dynamic, or a combination of the two. Abstract System Dependence Graphs (ASDG) [6] are a means of searching a graph of a system. Robillard [21] improved on this process by guiding the search based on an analysis of the system's topology. Instead of relying on structural information, other approaches make use of textual information found

in source code. The simplest approach is to search the code using a tool like *grep* to find relevant code using pattern matching, while more sophisticated techniques utilize Information Retrieval (IR) [17] and Natural Language Processing (NLP) [23]. Some approaches, such as SNIAFL [27] and DORA [14], incorporate both structural and textual analysis. There are also approaches that locate features dynamically. Software reconnaissance [24] discovers code related to a feature by analyzing two execution traces. Dynamic Feature Traces [12] improve upon this approach by including new selection criteria for trace scenarios and performing a more in-depth analysis.

Hybrid feature location leverages the benefits of both static and dynamic analyses. Eisenbarth et al. [11] apply Formal Concept Analysis to dynamic traces to produce a mapping of features to methods. Then, static dependencies are explored to locate additional code. In the PROMESIR approach [18], IR combines with a dynamic technique known as SPR [1] to rank methods likely relevant to a feature. In SITIR [16], a single execution trace is filtered using IR to extract code relevant to a feature. Static, textual, or dynamic analyses are all used to locate features in Cerberus [9].

Most existing coupling metrics capture coupling at the class level. Chidamber and Kemerer [7] introduced the measures Coupling Between Objects (*CBO*) and Response for a Class (*RFC*). Li and Henry also introduced a number of class coupling metrics such as Message Passing Coupling (*MPC*) and Data Abstraction Coupling (*DAC*). Briand et al. [5] built a framework for coupling measurement in object-oriented systems. Other static but non-structural coupling measures exist along semantic and evolutionary dimensions. The conceptual coupling metric, *CoCC* [19], captures a new dimension of coupling based on semantic data in source code. Interaction coupling [29] logical [13], and evolutionary [28] coupling utilize information from repositories to mine data from software artifacts that are frequently co-changed.

Instead of relying on static information to compute coupling, there are also dynamic class coupling metrics [2]. Dynamic analysis has been used to create the only existing feature coupling measure. Wong and Gokhale [25] defined the distance (*DIST*) between two features using an execution slice-based technique.

4. Proposed work

This research proposes to develop data fusion approaches for feature location and feature coupling. Data fusion involves incorporating multiple sources of data and considering each source to be an expert. The proposed feature location approach combines the expert opinions to better identify a feature's source code. The sources of information currently used are structural dependencies, textual information embedded in source code, and dynamic execution traces.

We have devised initial combinations of these three sources of information. Features can be found textually using Latent Semantic Indexing (LSI) [8]. Users formulate either a natural language query describing the feature or a query from the identifiers and comments of a known relevant method. LSI returns a list of all the methods in the system ranked by their similarity to the query. To combine textual and dynamic analyses, traces are collected for a feature, and then any methods not executed in the feature's traces are pruned from LSI's ranked list, thus reducing the number of irrelevant results. Two types of traces are used: full traces of an entire execution and marked traces in which the user can start and stop tracing at will to capture only the part of the execution when the feature is invoked. Finally, to combine textual, dynamic, and static information, a program dependency graph (PDG) is explored. Starting at a method known to be related to a feature, traverse dependencies based on textual and dynamic criteria. For example, if a method's neighbor in a PDG was not executed and does not meet a textual similarity threshold, then the dependency is not followed. Future work will include alternative configurations and the inclusion of other data mining techniques.

We use structural and textual information to capture feature coupling. Unlike *DIST*, the existing feature coupling metric, we avoid using dynamically-collected data because executing traces requires creating scenarios that invoke a feature in isolation. Creating such scenarios can be difficult. Therefore, our metrics rely on information available in a system's source code. We have developed five feature coupling metrics. Structural feature coupling (*SFC*), is defined as the percentage of methods shared by two features. A variant of this measure, SFC', also takes in to account the first order structural neighbors of a feature's methods. Textual feature coupling (*TFC*), is defined as the average of the textual similarities between all pairs of methods in both features. Its variant, TFC_{max}, only considers the strongest textual similarity between a feature's methods. *SFC* and *TFC* are combined by an affine transformation to define a Hybrid Feature Coupling measure (*HFC*).

5. Evaluation strategies

Performing a quantitative evaluation of different feature location techniques poses a difficult challenge because of the inherent uncertainty and subjectivity involved in finding all of the code that pertains to a feature. One programmer may think a method is relevant to a feature while another may not.

We intend to use at least three different methods to evaluate the various feature location techniques. First, we will use an artifact-based evaluation in which the methods that implement a feature are those methods that were modified in a change request related to that feature. This mapping provides a gold standard by which various measures such as precision and recall can be computed. For this type of evaluation, we will use large, open source systems like *Eclipse*.

A change request, such as a bug report, may pertain to only a small portion of a feature. As a result, the methods touched by a change request may not fully represent a feature. Therefore, we will also use a benchmark-based form of evaluation. Eaddy et al. [10] as well as Robillard et al. [22] have publicly available data sets that map features to code for several open source systems. We can use these data sets as benchmarks for our evaluation.

Notably, Eaddy et al.'s data sets were created by a single programmer, allowing room for subjectivity [10]. While Robillard et al.'s data sets come from several programmers, there is little agreement about which methods actually implement the features [22]. Therefore, as a third type of evaluation, we will use a top N strategy. Instead of asking programmers to search source code to find relevant methods, we will present them with the top N results of a feature location technique and ask them to determine the methods' relevance to a feature. This type of assessment should yield much higher agreement among programmers about which methods pertain to a feature.

By using three evaluation strategies that assess the problem from different perspectives, we hope to focus in on the best feature location approach. The idea is essentially data triangulation [26]: to synthesize data from multiple sources to come to a stronger conclusion.

To evaluate our feature coupling metrics, we need to know the methods that implement a system's features as well as the methods modified to fix bugs. For that reason, we will use Eaddy et al.'s [10] data sets again because they provide all the needed information. We will compute the Spearman rank order correlation coefficient between pairwise feature coupling and fault-proneness. This statistical test assesses the relationship between two variables, which in our case are the coupling between two features and defects.

Additionally, we can assess the potential of using feature coupling metrics for impact analysis. Using the measures of precision and recall, we can evaluate if the feature coupling metrics indeed indicate that coupled features tend to share defects. Precision is a measure of the exactness of the results; how many other features deemed coupled to a feature actually share a bug with it? Recall is the percentage of other features that share a bug with a feature that are coupled to a feature. To determine which features are coupled, a threshold is set, and we will vary this threshold to further explore the relationship between feature coupling and defects.

6. Preliminary results

In our previously published work [20], we evaluated ten different feature location techniques using the top N strategy. The two textual techniques used natural language queries and queries comprised of the identifiers and comments of a relevant method. The four approaches that combined textual and dynamic information utilized one of the two types of queries with either full or marked traces. Finally, textual, dynamic, and static information were used in tandem. Starting at a seed method, a program dependency graph is traversed based on textual and dynamic criteria to identify additional relevant methods.

Using four features each from the open source systems *jEdit* and *Eclipse*, the top ten results from each of these approaches were classified as either relevant, somewhat relevant, or not relevant to the feature of interest according to a set of guidelines adapted from [22]. The results were such that on average, only between one and three methods in the top ten pertained to the feature, somewhat falling short of the goal of finding near-complete implementations of features. However, we did observe that marked traces outperformed full traces. Another important finding was that method queries performed just as well as natural language queries provided by a user. This is a significant finding because method queries can be formulated automatically, removing the human input required to use natural language queries.

We also empirically evaluated our feature coupling metrics on two open source systems from Eaddy et al.'s publically available data sets: *dbViz* and *Rhino*. *dbViz* is a database visualization tool, and *Rhino* is a Javascript compiler. Both systems are written in Java. Using the mappings of features to code and bugs to features in Eaddy et al.'s data sets, we computed the Spearman rank order correlation coefficients between the coupling among pairs of features and bugs shared by those features. Using *SFC*, SFC′, *TFC*, TFC_{max}, and *HFC* (with a structural weight of 0.5 and a textual weight of 0.5), we found a moderate to strong statistically significant correlation between all our feature coupling metrics and defects. These findings suggest that feature coupling is a good predictor of fault-proneness. By comparison, the existing dynamic feature coupling measure, *DIST*, is not correlated with defects, and the results are not statistically significant.

Since class coupling has been used for impact analysis [4], we also investigated whether feature coupling metrics can support impact analysis by

computing precision and recall of the metrics at various thresholds. We found that *SFC* had the best precision but worst recall, and *TFC* had the best recall but worst precision. *HFC* falls in between these two.

7. Expected contributions

The proposed research will aid programmers as they perform feature-level software maintenance tasks. Our data fusion approach to feature location will help identify a feature's relevant code so a change task can be successfully initiated. Once the change is implemented, our feature coupling metrics can be used to determine if other features have been impacted by the modification. We expect this research to not only reduce the amount of time and effort that programmers spend working on maintenance tasks, but also to facilitate the development of higher quality software.

8. Acknowledgements

This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

9. References

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

[2] Arisholm, E., Briand, L. C., and Foyen, A., "Dynamic coupling measurement for object-oriented software", *Transactions on Software Engineering*, vol. 30, no. 8, August 2004, pp. 491-506.

[3] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in Proc. of International Conference on Software Engineering ,1994.

[4] Briand, L., Wust, J., and Louinis, H., "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", in Proc. of International Conference on Software Maintenance, 1999.

[5] Briand, L. C., Daly, J., and Wüst, J., "A Unified Framework for Coupling Measurement in Object Oriented Systems", *Transactions on Software Engineering*, vol. 25, no. 1, January 1999, pp. 91-121.

[6] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of International Workshop on Program Comprehension, 2000.

[7] Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design", *Transactions on Software Engineering*, vol. 20, no. 6, 1994, pp. 476-493.

[8] 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.

[9] 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 International Conference on Program Comprehension, 2008.

[10] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., and Aho, A. V., "Do Crosscutting Concerns Cause Defects?" *Transaction on Software Engineering*, vol. 34, no. 4, July-August 2008, pp. 497-515.

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

[12] Eisenberg, A. D. and De Volder, K., "Dynamic Feature Traces: Finding Features in Unfamiliar Code", in Proc. of International Conference on Software Maintenance, 2005.

[13] Gall, H., Jazayeri, M., Krajewski, J., "CVS Release History Data for Detecting Logical Couplings", in Proc. of International Workshop on Principles of Software Evolution, 2003.

[14] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of International Conference on Automated Software Engineering, 2007.

[15] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "On Computing the Canonical Features of Software Systems", in Proc. of Working Conference on Reverse Engineering, 2006.

[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 International Conference on Automated Software Engineering, 2007.

[17] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of Working Conference on Reverse Engineering, 2004.

[18] 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", *Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.

[19] Poshyvanyk, D. and Marcus, A., "The Conceptual Coupling Metrics for Object-Oriented Systems", in Proc. of International Conference on Software Maintenance, 2006.

[20] Revelle, M. and Poshyvanyk, D., "An Exploratory Study on Assessing Feature Location Techniques", in Proc. of International Conference on Program Comprehension, 2009.

[21] Robillard, M., "Automatic Generation of Suggestions for Program Investigation", in Proc. of Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2005.

[22] 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.

[23] 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 International Conference on Aspect Oriented Software Development 2007.

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

[25] Wong, W. E. and Gokhale, S., "Static and dynamic distance metrics for feature-based code analysis", *Journal of Systems and Software*, vol. 74, no. 3, February 2005, pp. 283-295.

[26] Yin, R. K., *Applications of Case Study Research*, 2 ed ed., CA, USA, Sage Publications, Inc, 2003.

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

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

[29] Zou, L., Godfrey, M. W., and Hassan, A. E., "Detecting Interaction Coupling from Task Interaction Histories", in Proc. of International Conference on Program Comprehension, 2007.