Using Information Retrieval to Support Design of Incremental Change of Software

Denys Poshyvanyk, Andrian Marcus*
Department of Computer Science
Wayne State University
Detroit, MI 48202
1-313-577-5408
[denys, amarcus]@wayne.edu

ABSTRACT

The proposed research defines an approach to combine Information Retrieval based analysis of the textual information embedded in software artifacts with program static and dynamic analysis techniques to support key activities of the incremental change of software, such as concept and feature location.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – enhancement, restructuring, reverse engineering, and reengineering

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Program understanding, feature identification, concept location, impact analysis, change propagation, dynamic and static analyses, information retrieval, coupling and cohesion measurement

1. PROBLEM DESCRIPTION

During the evolution of large scale software systems most activities involve making changes to the existing source code. Identifying the parts of the source code that correspond to a specific functionality is a prerequisite to program comprehension and is one of the most common activities undertaken by developers. This process is called *concept* (or *feature*) location and it is a part of the incremental change of software process [30].

Although incremental change ultimately needs to identify all components to be changed, the programmer must find the location in the code where the first change must be made. For that, the programmer uses a search process where the search space is the whole software and where diverse search techniques narrow down the search space. The literature limits this step to finding a small number of feature components. The full extent of the change is then handled by impact analysis, which is used to identify the remaining impacted components. In this research proposal, we are specifically addressing the identification of methods in object-oriented software that are part of the implementation of a feature (i.e., they change when the feature is altered) and can be used as a starting point in impact analysis.

While developers often perform feature location manually, tool support is needed for large and complex programs. Existing tools supporting feature location rely on data obtained via static and–or dynamic analysis of the program. While dynamic analyses often can not discriminate overlapping features, static analyses better filter and organize data, but they can rarely identify precisely elements of source code contributing to a specific execution scenario. The research community has long recognized the need to combine static and dynamic techniques [11] to improve the effectiveness of feature location [3, 9, 32, 36]. All these techniques are designed to be applied on the source code yet they do not capture important *textual* (or *lexical*) information which is embedded in identifiers and comments present in source code etc. Artefacts, such as call graphs or execution traces, generated from the source code provide in their structure information on *how* the system works, whereas textual artifacts capture information on *what* the system does, as well as important knowledge about the software domain, design decisions, developer information, communication, etc. We refer to these two types of information as *structural* and *semantic*, respectively.

In order to locate features and change a software system, developers must understand both *what* the system does and *how* it works, hence they need to analyze the two types of information. While these two types of information are complementary, there is *little support for their combination.* In particular, many of the existing tools *do not provide explicit representation for the semantic information*, but rather assume the implicit representation embedded in the textual software artifacts.

2. RESEARCH GOALS

We propose the use of Information Retrieval (IR) techniques to *extract and represent the semantic information in large scale software systems such that it can be automatically combined with structural information* to better support concept and feature location in source code. Specifically, the research will focus on combining IR-based analysis data with the analysis of program dependencies, execution traces to define new techniques for feature location.

We expect that these new techniques will contribute directly to improvement of design of incremental change and thus increased software quality and reduction of software maintenance costs.

* Dissertation advisor

3. RELATED WORK

Existing techniques for feature location broadly fall into three categories, based on the type of information they use: dynamic, static, and hybrid.

Software reconnaissance by Wilde et al. [38] was the early dynamic technique to identify features by analyzing execution traces of test cases. Two sets of test cases are used to obtain two execution traces: an execution trace where the desired feature is exercised and an execution trace where the feature is not exercised. The two traces are compared to identify the entities of the program that implement the feature. This technique was recently extended to improve its accuracy by introducing new criteria on selecting execution scenarios [10] and by analyzing the execution traces differently [3]. Similarly, Wong et al. [39] analyzed execution slices of test cases to identify features in the source code. Eisenbarth et al. [9] combined both static (i.e., dependencies) and dynamic (i.e., execution traces) data to identify features in programs and identify relations among them using Formal Concept Analysis.

A number of dynamic approaches exist, which use single scenario per feature. They are different from the previous approaches as they focus on identifying multiple features at a time or relationships among them. In particular, these approaches focus on feature interactions [8, 32], feature evolution [13], hidden dependencies among features [12] as well as identifying a canonical set of features for a given software system [16].

Biggerstaff et al. [4] introduced static feature location as the "concept assignment problem" and designed a tool that utilizes parsing, simple clustering, identifier names, and a browser, to support the identification. The simplest and most used static techniques are based on searching the source code using text pattern matching tools, such as Unix *grep* [1]. A significant improvement over the *grep*-based tools are the IR-based approaches [22], which provide ranked results to the developer's queries. AspectBrowser [14] improves searching experiences by searching for regular expressions and displaying the results graphically in programs visualized using map metaphors. Chen and Rajlich [5] proposed a technique for feature location based on searching the Abstract System Dependence Graph (ASDG). This process is improved in [31], where the search of the dependency graph is guided based on the analysis of the topology of the structural dependencies. Some methods combine different kinds of static information (i.e., semantic and structural), such as the one proposed by Zhao et al. [41], which uses Information Retrieval and a branch-reserving call graph to search the source code. The software reflexion model [23] is another technique that can be used in the context of concept location (assignment) to identify mappings between domain level concepts and their implementations (reflexions) in source code. Recently, Natural Language Processing (NLP) techniques were applied to support concept location in source code [34].

A comparison of different approaches for feature location in legacy systems is presented in [37]. A more up-to-date summary of all existing approaches can be found in [21], whereas a summary of industrial tools available for feature location is available in [35].

4. PROPOSED WORK

In the proposed work we utilize an Information Retrieval method, Latent Semantic Indexing (LSI) [7], as a text indexing and search engine. LSI is based on a Singular Value Decomposition (SVD) [33] of the co-occurrence matrix of identifiers and comments in source code documents of a software system. SVD is a form of factor analysis, which is used to reduce dimensionality of the feature space to capture most essential semantic information.

Originally LSI has been mostly applied on natural language corpora, however, the method has been shown to lend itself well on other types of data, for example, textual information extracted from source code and associated documentation. Some of the software engineering problems, related to concept location, which have been addressed using LSI are traceability link recovery between source code and documentation [2, 6, 19], requirements tracing [15], reverse engineering [17], high-level concept clones identification [18], conceptual cohesion [20] and coupling [25] measurement etc.

In order to index the source code with LSI and combine semantic information with other structural artifacts, the following set of steps is applied:

1. **Generating a corpus of a software system.** The source code is parsed using a developer-specified granularity level (that is, methods or classes) and documents are extracted from the source code. A corpus is constructed, so that each method (and/or class) will have a corresponding document in the corpus. Only identifiers and comments are extracted (and pre-processed) from the source code.
2. **Indexing software.** The corpus is indexed using LSI and a representation of the corpus as a real-valued vector subspace is obtained. Dimensionality reduction is performed in this step, capturing essential semantic information about identifiers, comments and their relationships in the source code. In the resulting subspace, each document (method or class) has a corresponding vector.
3. **Formulating a query.** A developer selects a set of terms that describe the concept of interest (for example, 'print page both sides'). This set of words constitutes the initial query. The tool spell-checks all the words from the query using the vocabulary of the source code (produced by LSI). If any word from the query is not present in the vocabulary, then the tool suggests similar words based on editing distance (or semantic similarities among words) and removes the term from the search query.
4. **Ranking documents.** Similarities between the user query and documents from the source code (for example, methods or classes) are computed. The similarity between a query reflecting a concept and a set of data about the source code indexed via LSI allows generating a ranking of documents relevant to the feature. All the documents are ranked by the similarity measure in descending order.
5. **Combining with static and dynamic analysis.** The developer formulates a *scenario* that captures the feature of interest; she marks the intervals in this scenario for which the trace should be collected and executes the program according to this scenario. A set of executed methods is obtained. If the user is uncertain on where to mark the traces, complete scenarios can be executed. Based on the LSI

index, obtained in step 4, the set of executed methods are sorted based on the similarity among the methods and the user query. In addition to the representation as a ranked list, the ranked executed methods can be presented with dependencies among them (i.e., based on call-graph relations).

6. **Presenting search results as a ranked list**. The programmer inspects the methods ranked in step 5, starting with the method on the top of the list. The developer may return to step 3 or 5 at any moment to reformulate a query or refine the execution scenario.
7. **Clustering search results**. This step aims at reducing developer's search effort by providing additional structure among the search results, such that parts of source code and documentation are automatically grouped and labeled based on common topics, similarly as it is done in web searching clustering engines such as Vivisimo[1] and Clusty[2].

5. EVALUATION STRATEGIES

One of the goals of the case studies is to allow for quantitative evaluation of different techniques. This is a notoriously difficult task as it is hard to define the entire extent of the implementation of a feature in large systems. One feature may be implemented by hundreds of methods and many of them may contribute towards several features. In order to have a gold standard against which we can define objective measures, we narrow the extent of feature implementation to those methods relevant to a change request.

Our techniques will be evaluated through case studies on large scale open source software systems such as Mozilla[3] and Eclipse[4]. The case studies will be done on both historical and current data of changes. Since the developers of these systems maintain detailed bug reports[5] and descriptions of respective modifications, the bug descriptions can be considered as change requests. Each documented bug can be used as a gold standard against which we compare the results of the feature location techniques, since the documentation of each bug specifies which methods were changed to fix that bug. We will consider changed methods as belonging to the feature associated with the bug. One method may belong to more than one bug (that is, changed in different bug fixes), but it is at least exercised in the associated feature.

6. PRELIMINARY RESULTS

Several publications resulted from this research [20, 24-29, 40]. In addition, the following prototype tools have been implemented to support our techniques: **I**nformation **R**etr**i**eval based **S**oftware **S**earch (IRiSS) [27], the Eclipse version of IRiSS [26] and **G**oogle **E**clipse **S**earch[6] (GES) [29]. Recently, we organized a working session[7] on IR-based approaches in software evolution to identify the main research trends and practical issues in the filed.

[1] http://vivisimo.com/
[2] http://clusty.com/
[3] http://www.mozilla.org/
[4] http://www.eclipse.org/
[5] https://bugzilla.mozilla.org/
[6] http://ges.sourceforge.net
[7] http://www.cs.wayne.edu/~amarcus/icsm2006/

In [24] we presented a novel feature location technique, namely **P**robabilistic **R**anking **O**f **M**ethods based on **E**xecution **S**cenarios and **I**nformation **R**etrieval (PROMESIR), formulated as a decision-making problem in the presence of uncertainty. The solution to the problem is formulated as a combination of expert opinions, whereas experts are represented by two existing techniques – **S**cenario-based **P**robabilistic **R**anking (SPR) of events based on processor emulation [3] and IR-based techniques that use LSI [22].

As both techniques provide rankings of source code elements in response to user input (SPR provides a ranked list of methods based on several execution scenarios containing/not containing the feature of interest; LSI ranks all methods according to a user query formulated as a set of terms present in the source code of a software system), we combine them as judgments of two independent experts, who provide expertise to solve the problem of identifying a feature precisely. The LSI expert builds its judgment based on the source code textual similarities, while SPR expert grounds its judgment on the probabilistic ranking of dynamic events observed in execution traces. However, both experts reply to the same question: "What are the locations of a feature of interest?". For the technical details on the combination of two methods refer to [24].

We empirically evaluated the combination of these techniques through several case studies, where we used the source code of several versions of Mozilla and Eclipse. We used PROMESIR to identify several features associated with several bugs in the source code of these systems. In response to every bug description report, which is used as a change request, two experts formulated a set of scenarios related to the bug and a set of queries containing descriptions of those bugs. In order to compare the methods, we computed accuracies for the PROMESIR, SPR and LSI. Overall, the results of the case studies showed that LSI and SPR, based on different analysis methods and data, complement each other, and the results obtained with the PROMESIR are significantly better than those of any of the techniques used independently [24].

Also we improved the existing IR-based technique for concept location [22] with automatic clustering of the search results using Formal Concept Analysis (FCA). The IR based concept location technique uses a search engine based on LSI, which allows the user to search source code and related textual documentation by writing natural language queries and retrieving a list of source code elements (for example, classes, methods, functions, files), ranked based on their similarity to the query. Based on the ranked results of the search the proposed approach will automatically generate a labeled concept lattice with search results prearranged into topics and categories. Developers can determine whether a node from the concept lattice (that is, topic or category) is relevant or not to their query by simply examining its label; they can then explore only relevant nodes in the lattice and ignore the other ones, thus reducing their search effort. We evaluated the novel approach in a case study on concept location in the source code of Eclipse. The results of the case study showed that the proposed approach is effective in organizing different concepts and their relationships present in the search results. The proposed concept location method outperforms the simple ranking of the search results, reducing the programmer's effort.

7. EXPECTED CONTRIBUTIONS

The proposed research will help obtain new insights into the design and implementation of incremental change of software via improved techniques for feature and concept location. It is expected that the resulting techniques, based on the combination of semantic (textual), structural and dynamic information, not only reduce programmers' effort for searching and changing software but also help improve software quality.

The set of tools that support the designed methods will be released to the research community and improved based on the feedback from various users.

8. REFERENCES

[1] Aho, A. V., "Pattern matching in strings", in *Formal Language Theory: Perspectives and Open Problems*, New York Academic Press, 1980, pp. 325-347.

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Trans. on Soft. Engineering*, vol.28,no. 10,Oct. 2002,pp.970-983.

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

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

[5] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. *ICPC'00*, pp. 241-249.

[6] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM Transactions on Software Engineering and Methodology,* 2007.

[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] Egyed, A., Binder, G., and Grunbacher, P., "STRADA: A Tool for Scenario-Based Feature-to-Code Trace Detection and Analysis", in Proc. of International Conf. on Software Engineering, 2007, pp. 41-42.

[9] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, 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 International Conf. on Software Maintenance, 2005, pp. 337-346.

[11] Ernst, M., "Static and Dynamic Analysis: Synergy and Duality", in Proc. of ICSE Workshop on Dynamic Analysis, 2003, pp. 24-27.

[12] Fischer, M., Pinzger, M., and Gall, H., "Analyzing and Relating Bug Report Data for Feature Tracking." in Proc. of 10th IEEE Working Conference on Reverse Engineering, 2003, pp. 90-101.

[13] Greevy, O., Ducasse, S., and Girba, T., "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", in Proc. of 21st IEEE International Conference on Software Maintenance, 2005, pp. 347-356.

[14] Griswold, W. G., Yuan, J. J., and Kato, Y., "Exploiting the Map Metaphor in a Tool for Software Evolution", in Proc. of 23rd IEEE International Conference on Software Engineering, 2001, pp. 265-274.

[15] Hayes, J. H., Dekhtyar, A., Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study of methods", *IEEE Transaction on Software Engineering*, vol. 32, no.1, January, pp.4-19.

[16] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "On Computing the Canonical Features of Software Systems", in *13th IEEE Working Conference on Reverse Engineering,* Benevento, Italy, 2006.

[17] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proc. of 23rd IEEE/ACM Int. Conf. on Software Engineering, 2001, pp. 103-112.

[18] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proc. of *ASE'01*, pp. 107-114.

[19] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, October 2005, pp. 811-836.

[20] Marcus, A. and Poshyvanyk, D., "The Conceptual Cohesion of Classes", in Proc. of *ICSM'05*, pp. 133-142.

[21] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in Proc. of 13th *IWPC'05*, pp. 33-42.

[22] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of 11th *WCRE'04*, pp. 214-223.

[23] Murphy, G. C., Notkin, D., and Sullivan, K. J., "Software Reflexion Models: Bridging the Gap between Design and Implementation", *IEEE Trans. on Software Engineering* vol. 27, no. 4, 2001, pp. 364-380.

[24] 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 Trans. on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.

[25] Poshyvanyk, D. and Marcus, A., "The Conceptual Coupling Metrics for Object-Oriented Systems", in Proc. of *ICSM'06*, pp. 469 - 478.

[26] Poshyvanyk, D., Marcus, A., and Dong, Y., "JIRiSS - an Eclipse plug-in for Source Code Exploration", in Proc.of *ICPC'06*, pp. 252-255.

[27] Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A., "IRiSS - A Source Code Exploration Tool", in Proc. of 21st IEEE International Conference on Software Maintenance, 2005, pp. 69-72.

[28] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of *ICPC'07*, pp. 37-48.

[29] Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D., "Source Code Exploration with Google ", in Proc. of 22nd IEEE International Conf. on Software Maintenance, 2006, pp. 334 - 338.

[30] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", in *IEEE Software*, 2004, pp. 2-9.

[31] Robillard, M., "Automatic Generation of Suggestions for Program Investigation", in Proc. of ACM SIGSOFT ESEC/FSE'05, pp. 11 - 20.

[32] Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M., "Scenario-driven dynamic analysis for comprehending large software systems", in Proc. of 10th IEEE European Conference on Software Maintenance and Reengineering, 2006, pp. 71-80.

[33] Salton, G. and McGill, M., *Introduction to Modern Information Retrieval*, McGraw-Hill, 1983.

[34] 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 AOSD'07, pp. 212-224.

[35] Simmons, S., Edwards, D., Wilde, N., Homan, J., and Groble, M., "Industrial tools for the feature location problem: an exploratory study", *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 6, 2006, pp. 457-474.

[36] Tonella, P. and Ceccato, M., "Aspect Mining through the Formal Concept Analysis of Execution Traces", in Proc. of 11th IEEE Working Conference on Reverse Engineering, 2004, pp. 112 - 121

[37] 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, Feb.2003, pp.105-114.

[38] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., "Locating User Functionality in Old Code", in Proc. of *ICSM'92*, pp. 200-205.

[39] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S., "Locating program features using execution slices", in Proc. of IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, 1999, pp. 194-203.

[40] Xie, X., Poshyvanyk, D., and Marcus, A., "3D Visualization for Concept Location in Source Code", in Proc. of *ICSE'06*, pp. 839-842.

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