A Weighted Call Graph Approach for Finding Relevant Components in Source Code

Shin-Young Ahn
S/W & Contents Convergence Technology Team
ETRI
Daejeon, Korea
syahn@etri.re.kr

Sungwon Kang, Jong-Moon Baik, Ho-Jin Choi
Software Technology Institute
ICU
Seoul, Korea
{kangsw, jbaik, hjchoi}@icu.ac.kr

Abstract—**To reuse open source code it is necessary to understand how the software is implemented and how the software architecture is designed. However, it is not an easy task because open source typically does not have the design document which maps features to source code components. In this paper, we propose a method to find the components relevant to a feature in open source code. Our solution is a static approach to recover the relationship between the software manual and the source code and it is based on information retrieval technique. In our approach, we build and analyze a weighted call graph using the similarity values obtained from the vector space information retrieval. Experimental application shows that our method is more robust and is less influenced by cut level than the method of Zhao et al.[3].**

Software Understanding; Open Source; Requirement; Feature; Recovering Traceability

I. INTRODUCTION

Reusing existing open sources is an efficient way of developing software under limited time and budget. However, understanding the open sources program has been a big challenge because open sources lack documentation. The reason why open sources generally have no design documentation is that open sources are developed by many developers and are revised by other people over time. Typically, open sources tend to provide a brief manual which explain the features of the open sources program. Therefore, effective method of understanding open sources is a crucial factor for successful reuse of open source.

There are two major approaches for reusing open source [1]. One is a systematic approach in which developers try to understand the program behaviors completely before any code modification. The other approach is to understand the program partially so as to locate as quickly as possible certain code segments that should be changed. If the scope of modification is small, the need-based approach is preferred because the systematic approach may take more time and effort [2,4]. From the viewpoint of maintenance, partial understanding means locating the components of source code that correspond to a specific program feature (i.e. a functional requirement) [2,3,4,5,6].

The top-down strategy has been most frequently used for understanding source code. In this strategy, an analyst constructs a call graph with nodes representing called functions. The analyst recursively visits the called functions until some relevant code is found. This traversal uses a depth-first search algorithm and visits half of the nodes of a given call graph on average. In the worst case, the target node may be visited last.

The goal of our work is to provide a top-down method of discovering code segments in which some features are implemented. Our idea is to use the relation between a given manual (we regard manuals as requirements specification because manuals include the features of the open sources system) and its source code. Generally, the verbs and nouns in the manual tend to be used in the source code as identifiers or words in comments. Such identifiers and comments provide the relationship between the manual and the source code. To recover relationship, we use the vector space information retrieval (IR) technology. We can extract certain keywords from the manual and retrieve functions with the keywords via IR. Because IR can't guarantee finding correct answers by itself, we propose using a weighted call graph that helps us select core functions among the retrieved functions. Then by analyzing this weighted call graph, we determine the final set of relevant functions.

The remainder of this paper is organized as follows. In chapter II, we discuss the related work. In chapter III, we present our method to discovering relevant functions in source code. Next, Chapter IV shows an experimental study and compares our method with the previous work. Finally, chapter V discusses our work.

II. RELATED WORKS

The methods to find code components relevant to a feature can be classified as: 1) dynamic methods, 2) static methods, and 3) hybrid methods that combine dynamic and static methods.

Dynamic method is to understand a program by running it. Wilde et al. [7,18] proposed Software Reconnaissance based on test suite execution to locate features. Wong et al. [4] proposed an execution slice-based solution, where execution slices are the set of program code executed by a test case. This method is the best one when a feature is controllable by input data [10]. Eisenberg et al [17] introduced an automated technique for feature location: Dynamic Feature Traces (DFTs) based on execution-trace analysis. Licata et al. [20] uses the user test suites in different versions to define Feature Signature which measure the evolution of a program. This approach is useful to understand how a system changes over time. Dynamic methods require recompiling source code with debug option or modifying source code so that each function prints a visiting-log whenever it is visited. However, the method to modify source code takes much time and recompiling makes it hard to analyze the execution slices. Dynamic methods may consume excessive time and file space rather than static method [4].

Static method is to analyze a program without execution. Chen and Rajlich [2] are pioneers in static approach for locating features. They proposed a computer-assisted search process to find out relevant code segments by traversing Abstract System Dependence Graph (ASDG) extracted with the program analyzer. Antoniol et al. [8] tried to recover traceability between requirements and source code using IR without additional source code analysis. Marcus et al. [19] address the concept location using Latent Semantic Indexing (LSI) retrieval method. Zhao et al. [3] combined information retrieval technique and static analysis of source code structure. First, they used IR to retrieve initial function set related with a specific requirement. Then they complement the initial retrieved function set by static call chain analysis.

Eisenbarth et al. [6,9] proposed a hybrid approach using Concept Analysis. They introduced a semi-automatic technique, a process analyzing the relation among features, scenarios, and computational units. This method is not suitable for features that are only internally visible.

III. OUR APPROACH

The IR method is known to be appropriate for locating features in source code [3,8,13]. To improve the existing IR method, we augment the IR method with the construction and the analysis of a weighted call graph. We call our approach Weighted Information Retrieval (WIR). WIR uses IR to get each function's similarity weight information and core function, which is similar to central function in the Zhao's paper [3]. The main idea of WIR is to perform structural analysis of weighted call graph that is decorated with relevance information for a functional requirement. A weighted call graph can guide the analyst for efficient traversal of the call graph because it provides priorities among the paths to visit. We present WIR first by explaining the concept of weighted call graph and then presenting the actual process of WIR.

A. *Weighted Call Graph*

A weighted call graph is a specific call graph with weights at nodes and edges. *Weight* indicates the degree to which a node or an edge in the call graph might be related with a specific feature. We use similarity value between feature and source code as weight value for weighted call graph. The similarity is derived from the vector space IR. The reason why we use this similarity as weight is because the developer generally uses meaningful words from the user manual (or requirements specification) as names for program items, such as functions, variables, types, classes, methods and comments when they write source code[8]. The similarity gained from IR has also proved trustworthy in many areas [8]. The weight of a node is called *node weight* (NW) and the weight of an edge is called *edge weight* (EW). Both NW and EW consist of a pair of weight data: *relevance weight* (RW) and *irrelevance weight* (IW).

A node RW is similarity value for the requirement of interest. A node IW is selected as the maximum value among the function's similarity values for the uninteresting requirements. The reason why we do not add up all the irrelevance values for an uninteresting requirement is because the sum of irrelevance values can not specify explicitly how much functions are unrelated. EW is calculated based on NW. An edge RW is the sum of the RWs of the children nodes. An edge IW is the sum of the IWs of the children nodes.

B. *The WIR Process*

The process of the WIR method consists of six steps as shown in Figure 1. Step 1, 2 and 3 are related to IR. Step 4 and 5 are related to the construction of a weighted call graph. In step 6, the final relevant function set is selected through analysis of weighted call graph.

Step 1: Prepare retrieval. In this step, the analyst manually transforms manual and source code to query document and function document respectively that IR engine can read and understand. Query document is composed of text paragraphs for each functional requirement. Function document comprises of a block of code describing a function. Each paragraph and each block include a delimiter and a sequential number.

Step 2: Index query and function document. A paragraph of query document is transformed into a set of index terms by the IR engine. The nouns and verbs in the paragraph are extracted. A block of code in the function document is also indexed by the IR engine. Extracted identifiers include function names, variable names, comments, and names of the functions invoked in a function. Identifiers composed of two or more words are separated. Then the processed identifiers are transformed into a set of index terms using the text normalization.

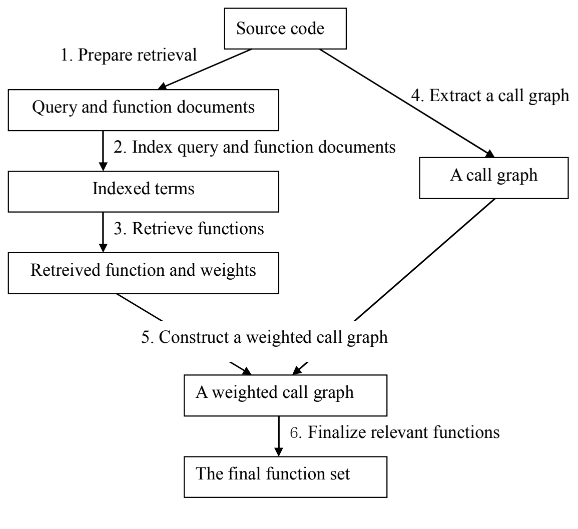
Source code
1. Prepare retrieval
4. Extract a call graph
Query and function documents
2. Index query and function documents
A call graph
Indexed terms
3. Retrieve functions
Retreived function and weights
5. Construct a weighted call graph
A weighted call graph
6. Finalize relevant functions
The final function set

Figure 1. The WIR process

Step 3: Retrieve functions. The vector space IR model regards functions and queries as vectors. In Step 2, index terms for queries and functions were prepared. From the index terms, the query and function vectors are calculated. According to the equations of vector space IR model, the retrieval process for one query in the query document is performed as follows: The index terms are components of certain vector which express each query and document. The meaning of inverse document factor is that the more frequently index term occurs in all functions, the less important role the index term plays in the retrieval process. The normalized term frequency factor of each index term and the inverse document factor of each index term are used to calculate the weight of function vector and query vectors .

Step 4: Extract a call graph. In WIR, the call graph is used as such a higher level model. The call graph extractor analyzes a collection of source file and outputs a graph depicting the dependencies between various function calls. The call graph begins with the main function (i.e. main()) and is expanded recursively through the nodes called by each function node until all functions are visited.

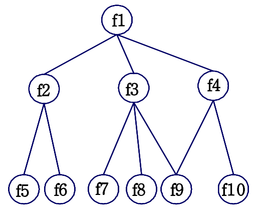

Figure 2. An example of a call graph

Step 5: Construct a weighted call graph. This step is invoked when a retrieved function with weight information and a call graph are provided as input. In the call graph in Figure 2, there are 10 nodes and 10 edges. A weighted call graph is obtained from an (ordinary) call graph by adding each node with RW and IW, which are generated via IR for a requirement of interest and uninteresting requirements.

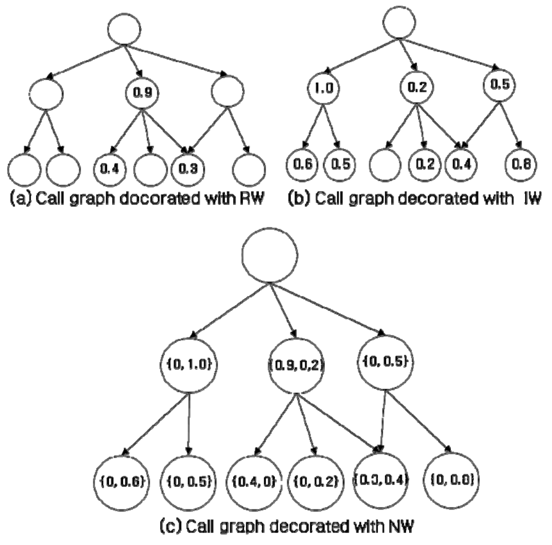

Figure 3. Calculating NWs for a call graph

We will show an example. First, we assume a set of RW and a set of IW which merge the sets of IW for irrelevant requirements.

- RW = { (f3, 0.9), (f7, 0.4), (f9, 0.3)}
- IW = { (f2, 1.0), (f3, 0.2), (f4, 0.5), (f5, 0.6), (f6, 0.5), (f8, 0.2), (f9, 0.4), (f10, 0.8) }

Second, RW and IW are merged into NW. NW maps a function node into a pair of weights <RW, IW>.

- NW = {(f2, <0,1.0>), (f3, <0.9,0,2>), (f4, <0, 0.5>), (f5, <0,0.6>), (f6, <0,0.5>), (f7, <0.4,0>), (f8, <0,0.2>), (f9, <0.3,0.4>), (f10, <0,0.8>)}

Third, EW is calculated as follows. EW of an edge is the sum of the weights of nodes that can be visited through the edge. Let *E* be an edge. Let *N* be a node reachable directly via *E*. Let E_k be an edge connected to *N*. Let the weight function be *W*( ). The weight of edge *E* is calculated from the next equation:

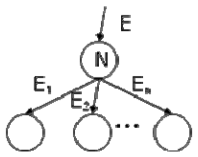

Figure 4. Calculating EW for an edge

$$W(E) = W(N) + \sum_{k=1}^{n} W(E_k) \quad (1)$$

where *n* is the number of edges connected to *N*. So the edge weight should be calculated from the bottom to the top recursively.

- EW = {(f1→f2,<0,2.1>), (f1→f3,<1.6,0.8>), (f1→f4,<0.3,1.7>), (f2→f5,<0,0.6>), (f2→f6,<0, 0.5>), (f3→f7,<0.4,0>), (f3→f8,<0,0.2>), (f3→f9,<0.3,0.4>), (f4→f9,<0.3,0.4>), (f4→f10,<0, 0.8>)}

The resulting weighted call graph is in Figure 5.

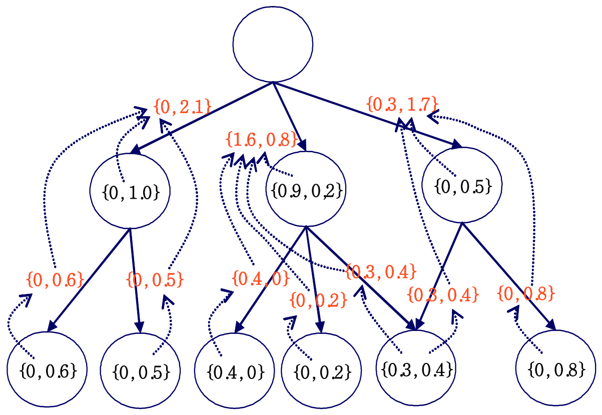

Figure 5. Calculating EWs in a call graph

Step 6: Finalize relevant functions. Finally, a set of functions relevant to a specific feature is selected. This step contains three stages. The first stage is selecting core function, the second is further retrieve functions in the final set and the third stage is weight analysis of the given call graph.

The core function is selected among the functions retrieved via IR. The core function has higher weight and lower reference. The core function may not have the highest RW because the criteria of selecting core function are the more relevant to a specific requirement and the less relevant to the other requirements. Even if a function is top ranked in RW, if its IW is higher than its RW and it is related to many uninteresting feature, then it could not be a core function. That is to say, the function related to many requirements may not be a core function for any requirement. Core function analysis proceeds by selecting call chains that includes a core function. Through this analysis, all the functions between the main function and the core function are added to the final set and all the functions which are called by core function recursively are also added to the final set.

Weight analysis selects a call chain by recursively visiting edges with highest weights from the root (main function) to the leaves. During visiting function nodes through the edges with the highest weight, when all subsequent edges of the node have lower RWs than the RW of the highest ranked function, then all the nodes visited through these subsequent edges are selected as a final set.

IV. An Experimental Application

The target system for our experiment is an application system from GNU with the name DC [14]. The same application was used in [3]. DC is distributed with the BC package. The DC system was published by the Free Software Foundation. DC system contains complete open source code and documentation that describe the detailed functional requirements expressed in English. DC system consists of 49 functional requirements and 74 user functions.

A. Procedure and Tools used for the Experiment

Before conducting the experiment we analyzed DC system and got the set of relevant functions related with each functional requirement. This set was saved in a file. The experimental environment is Linux operating system which supports GNU compiling because the target system was written in C. To apply our approach, we used two tools: SMART and modified cflow.

The first step of WIR is to get similarity information for functions that are relevant to a specific feature. SMART [15,16] was used for measuring similarity between requirement and source code. It is an implementation of vector space model IR proposed by Salton. This tool outputs the retrieved function set ranked by similarity. These functions and similarity values are passed to the second tool.

The second tool is cflow [12] published by Free Software Foundation. The cflow utility analyzes a collection of source files written in C and outputs a graph charting dependencies between various functions. We modified cflow for our approach as follows: First, the modified cflow generates a reduced call-graph that only includes the user-defined functions. Second, it reads relevant weight and irrelevance weight information from the output of SMART: the tools get the number of interesting requirement from the command line and read the result of IR saved in a file. Third, it makes a weighted call graph: The weight information is saved into the each function node's data field. After that, edge weight information is calculated from the node weight data. Fourth, it chooses the final set that is related with a specific functional requirement: the tool adds all function nodes visited by core function analysis and weight analysis into the final set. Finally, it calculate precision and recall: the tool reads the relevant function set and checks the relevant functions in the final set. It finally calculates precision and recall.

B. Analysis of the Experiment Results

When the WIR process is over, we have two groups of function sets. The first group is the set of relevant functions that is obtained by manual analysis of the target system. The second group is the final function set generated from WIR process. We compare these two groups to evaluate the soundness and completeness of our approach.

For evaluation of our result, we used two metrics, *precision* and *recall*, which are commonly used metrics in the IR field. *Precision* measures the soundness of our approach. It is the ratio of the relevant documents retrieved (the set Ra) to the retrieved documents (the set A):

$$precision = \frac{|Ra|}{|A|} \tag{2}$$

Recall measures the completeness of our approach. It is the ratio of the relevant documents retrieved (the set Ra) to the relevant documents (R):

$$recall = \frac{|Ra|}{|R|} \tag{3}$$

We used a cut level N to select the first N functions according to rank value in the retrieved function set via IR. We experimented for cut levels 3, 6, 9, 12 and 15 to compare our result with that of [3]. To calculate the average result, we performed the experiment for all the requirements of the DC system. Table I shows the result.

As shown in Table I, *precision* decreases as the cut level increases. The reason is that as more retrieved functions are considered, more irrelevant functions tend to be included in the final set.

TABLE I. Comparison of the WIR method and the Zhao's approach

Metric \ Cut level		***3***	***6***	***9***	***12***	***15***
precision	***Zhao' approach***	82.10	69.70	61.20	59.37	56.11
	WIR method	69.34	64.50	60.85	59.10	56.28
recall	***Zhao' Approach***	43.24	56.76	62.58	69.43	72.35
	WIR method	79.70	82.50	83.32	84.59	85.08

WIR does not accomplish higher precision than Zhao's method when cut level is low. It is because WIR adds more functions into the final set through weighted call graph analysis. However, as the cut level becomes higher, the precision of WIR becomes higher than Zhao's because it needs more weight information to build weighted call graph.

On the other hand, *recall* increases as the cut level increases as shown in Figure 8 and Table 1. This is because as more retrieved functions are considered, more relevant functions tend to be included in the final set. *Recall* does not care about the number of retrieved set but cares about how many relevant functions are included in the final set over the relevant function set.

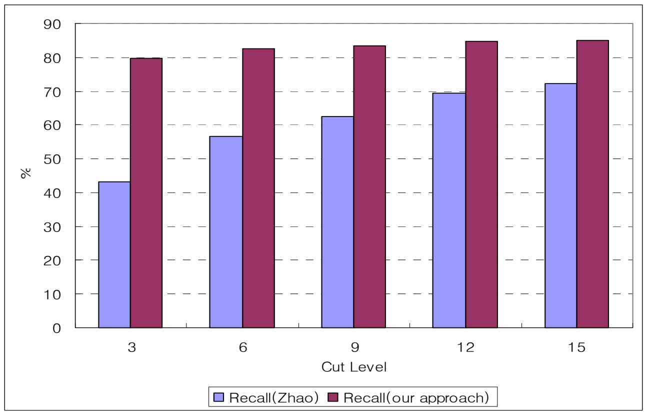

Figure 6. Comparison of recalls

WIR accomplishes a higher recall value than Zhao's method. When the cut level is 3, WIR gets 79.7% recall. This value is 36.46% higher than Zhao's. When cut level is 15, WIR gets 85.08% recall, which is 12.73% higher than Zhao's. Whereas the variance of Zhao's between the cut levels 3 and 5 is 29.11%, the variance of WIR is only 5.38%. This shows that WIR is more robust than Zhao's method and, moreover, less influenced by cut level.

V. CONCLUSION

In this paper, we propose a static method for finding functions related to a feature. Our approach combines two analysis methods: the core function analysis and the weight analysis based on the weighted call graph. Our approach overcomes the imprecision and incompleteness of the exiting information retrieval (IR) method. The existing method shows the best result when the words in the feature description are used as identifiers in the source code. However, the method may cause negative effect when the feature description contains some trivial or general words. When these trivial words are used as identifiers in the source code, then irrelevant functions might be selected as core functions. Eventually, irrelevant functions are included in the final set through core function analysis. This situation decreases the precision and recall. Our weight analysis can be a solution of this situation by selecting the function nodes on the highest weight path in the weighted call graph. Our approach improves completeness when the function ranked highest by IR is not a relevant function and the feature of interest is implemented in several functions separately that have lower similarity value rather than the highest ranked function. The experiment result shows that our approach has a much higher recall than the existing method.

Our method might have some limitations. This method needs source code and feature description, and these must be preprocessed by human. Although our approach finds some components for a feature, we do not expect that the components are promptly reused in another system. The goal of our approach is to reuse the system by partial modification. If the components are to be reused in another system, the components should be modified by wrapping it for the new system. Moreover, our method is not a perfect solution because it does not find the exact set of relevant components but a set of probably relevant components.

Our approach is an effective method to help understand open source code partially. It is also useful to the developer who wants to modify a specific feature of an open source system. Our method saves time to understand and analyze open source code by providing the developer the relevant components of a necessary feature. Our future work will be enhancing the precision of finding relevant components. In this work, we used a vector space model for information retrieval. The vector space model is a basic retrieval model. We will be able to use another retrieval method based on semantic discovery.

REFERENCES

[1] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, *Mental Models and Software Maintenance in Empirical Studies of Programmer*, Ablex publishing Corp., Norwood, NJ, 1986.

[2] K. Chen, and V. Rajlich, "Case study of feature location using dependence graph" Proc. 8th Int'l Workshop on Program Comprehension, pp. 241 – 247, 2000.

[3] W. Zhao, L. Zhang, Y. L. Jing, and L. J. Sun, "Understanding how the requirements are implemented in source code," The 10th Asia-Pacific Software Engineering Conf, pp. 68 – 77, 2003.

[4] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi, "Locating program features using execution slices," Proc. 1999 IEEE Symp. Application-Specific Systems and Software Engineering and Technology, pp. 194 – 203, March 1999.

[5] N.Wilde, and C. Casey, "Early field experience with the Software Reconnaissance technique for program comprehension," Proc. Int'l Conf. on Software Maintenance, pp. 312 – 318, 1996.

[6] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," IEEE Trans. on Software Engineering, Vol. 29-3, pp. 210 – 224, March 2003.

[7] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," Proc. Conf. on Software Maintenance, pp. 200 – 205, Nov. 1992.

[8] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," IEEE Trans. on Software Engineering, Vol. 28-10, pp. 970 – 983, Oct. 2002.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," Proc. IEEE Int'l Conf. on Software Maintenance, pp. 602 – 611, Nov. 2001.

[10] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A comparison of methods for locating features in legacy software," Journal of Systems and Software 65, pp. 105 – 114, 2003.

[11] T. R. Madanmohan, and De' Rahul, “Open source reuse in commercial firms,” IEEE Software Vol. 21-6, pp. 62 – 69, 2004.

[12] GNU, “cflow online manual,” http://www.gnu.org /software/cflow/manual/index.html

[13] R. Baeza, and B. Ribeiro, *Modern Information Retrieval*, Addison Wesley, 1999.

[14] GNU, “DC: An Arbitrary Precision Calculator,” http://directory.fsf.org/GNU/bc.html

[15] G. Salton, *The SMART Retrieval System – Experiments in Automatic Document Processing*, Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[16] “SMART package,” ftp://ftp.cs.cornell.edu/pub/smart/ smart.11.0.tar.Z

[17] A. D. Eisenberg, and K. De Volder, “Dynamic Feature Traces: Finding Features in Unfamiliar Code”, Proc. IEEE Int’l Conf. on Software Maintenance, pp. 337-346, 2005

[18] N. Wilde, M. Scully, Software reconnaissance: Mapping features to code. Software Maintenace: Research and Practice, 7(1):49-62, 1995

[19] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic, “An Information Retrieval Approach to Concept Location in Source Code”, Proc. of the 11th Working Conference on Reverse Engineering(WCRE’04)

[20] D. R. Licata, C. D. Harris, S. Krishnamurthi, “The feature signatures of evolving programs”, Proc. of the 18th Intl Conf. on Automated Software Engineering, pp. 281-286, IEEE Computer Society, Oct. 2003