

Understanding How the Requirements Are Implemented in Source Code

Wei Zhao, Lu Zhang, Yin Liu¹, Jing Luo, Jiasu Sun
Software Engineering Institute, Peking University, Beijing, P. R. China, 100871
{zhaow, zhanglu, luo, sjs}@sei.pku.edu.cn
¹mirainmay@hotmail.com

Abstract

For software maintenance and evolution, a common problem is to understand how each requirement is implemented in the source code. The basic solution of this problem is to find the fragment of source code that is corresponding to the implementation of each requirement. This can be viewed as a requirement-slicing problem -- slicing the source code according to each individual requirement. In this paper, we present an approach to find the set of functions that is corresponding to each requirement. The main idea of our method is to combine the information retrieval technology with the static analysis of source code structures. First, we retrieve the initial function sets through some information retrieval model using functional requirements as the queries and identifier information (such as function names, parameter names, variable names etc.) of functions in the source code as target documents. Then we complement each retrieved initial function set by analyzing the call graph extracted from the source code. A premise of our approach is that programmers should use meaningful names as identifiers. Furthermore, we perform an experimental study based on a GNU system. We use two basic metrics: precision and recall (which are the common practice in the information retrieval field), to evaluate our approach. We also compare the results directly acquired from information retrieval with those that are complemented through static source code structure analysis.

Keywords: program comprehension, software maintenance, traceability, information retrieval, static analysis.

1. Introduction

For a maintainer of a legacy system, the first major problem he or she is facing may be to understand the software system. The understanding of a software system contains two main tasks: i) to understand what the software system does and ii) how it is implemented.

Usually, the first task is much easier than the second one, because this knowledge can be acquired from domain experts or the documentation of the software system. Although the second task can be done on the basis of the first task, there is an essential difficulty for the second task, because it is often quite difficult to match the human-oriented knowledge acquired from domain experts or the documentation against the program-oriented knowledge acquired from source code. Therefore, except some research on the concept assignment problem (such as [6]), the majority researchers are focusing on abstract representations of source code that may help maintainers to have the general knowledge of the program structure (see e.g. [7] and [15]).

In this paper, we view the second task as understanding how requirements are implemented in source code. The precondition is that there exists requirements documentation containing the human-oriented knowledge. In this paper, we call it the requirement-slicing problem -- slicing the source code according to each individual requirement. When the requirements need to be updated, it is obviously very important for maintainers to know which functions are associated with each requirement. Naturally, the requirement in this paper refers to the functional requirement.

Our solution is a static approach, which combines the recovery of the traceability between the words in the requirements documentation and the identifiers used in the source code with the static analysis of the source code structure. The premise of the traceability is that programmers should use meaningful names as identifiers. That is to say, the identifiers named by human-oriented concepts in source code can ultimately keep the traceability with the documentation. In our approach, we use the information retrieval (IR) technology to recover this hidden traceability. We treat requirements as the queries in the IR field and functions as the documents. Because of the intrinsic imprecision and incompleteness of the IR technology, we complement the initially retrieved function set through the analysis of the call

graph extracted from the source code to determine the final set.

The organization of the remaining of this paper is as follows. In section 2, we give the background knowledge about the vector space model in the information retrieval technology. Section 3 presents our approach to locate the function set corresponding to each requirement. In section 4, we give an experimental study of our approach. Section 5 discusses some related work and section 6 concludes this paper.

2. Background

In our approach, there are two main parts: the information retrieval part and the static analysis of source code structure part. In the information retrieval part, we apply the vector space model for indexing documents and queries and ranking the results. We introduce the vector space model in brief in this section. Please refer to [5] for details.

The vector space model [16], [17] proposes a framework in which partial matching is possible. It treats queries and documents as vectors constructed by the index terms. The index terms are acquired from the text of queries and documents according to some rules (such as ignoring articles, punctuations, numbers, etc.). Each index term has different weights in different document and query vectors. These term weights are ultimately used to compute the degree of similarity between each document and each query. Vector $(w_{1,q}, w_{2,q}, \dots, w_{t,q})$ represents the query q , in which $w_{i,q}$ is the weight of the i th index term in the query q , and t is the number of the index terms. Vector $(w_{1,j}, w_{2,j}, \dots, w_{t,j})$ represents the document d_j , in which $w_{i,j}$ is the weight of the i th index term in the document d_j , and t is the number of the index terms. The vector space model proposes to evaluate the degree of similarity of the document d_j with regard to the query q as the correlation. This correlation can be quantified by the cosine of the angle between these two vectors, which is shown in equation (1).

$$\text{sim}(d_j, q) = \frac{\bar{d}_j \cdot \bar{q}}{|\bar{d}_j| \times |\bar{q}|} = \frac{\sum_{i=1}^t w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^t w_{i,j}^2} \times \sqrt{\sum_{i=1}^t w_{i,q}^2}} \quad (1)$$

In order to compute the degree of similarity using equation (1), we need to specify how the index term weights are obtained. In the vector space model, the *tf* (term frequency) factor and the *idf* (inverse document frequency) factor are applied to decide the weights of index terms. The computation of these two factors is shown in equations (2) and (3).

$$f_{i,j} = \frac{\text{freq}_{i,j}}{\max_l \text{freq}_{l,j}} \quad (2)$$

In equation (2), $\text{freq}_{i,j}$ is the raw frequency of the i th index term in the document d_j (i.e., the number of times the i th index term is mentioned in the text of the document d_j); the maximum is computed over all index terms which are mentioned in the text of the document d_j ; and $f_{i,j}$ is the normalized frequency of the i th index term in document d_j .

The computation of inverse document frequency for the i th index term, idf_i , is given by

$$\text{idf}_i = \log \frac{N}{n_i} \quad (3),$$

where N is the total number of documents in the system and n_i is the number of documents in which the i th index term appears. The motivation for usage of the *idf* factor is that terms that appear in many documents are not very useful for distinguishing a relevant document from a non-relevant one.

Then, as suggested in [5], the two following equations can be used to compute the weights of index terms in documents and queries.

$$w_{i,j} = f_{i,j} \times \text{idf}_i \quad (4)$$

$$w_{i,q} = \left(0.5 + \frac{0.5 \text{freq}_{i,q}}{\max_l \text{freq}_{l,q}}\right) \times \text{idf}_i \quad (5)$$

3. The Approach

The basic idea of our approach is combining IR method with program structure analysis to locate the function set according to certain requirement. As a requirement usually contains some information of its implementation, we can use the IR technology to retrieve the functions that are related to the implementation information. These functions can be assumed as the functions that will be invoked when executing the requirement. Furthermore, call graph extracted from the source code is used to complement the results obtained through information retrieval to solve the intrinsic imprecision and incompleteness of the information retrieval technology.

In the information retrieval part, we use each functional requirement acquired from the documentation of the software system as the query to retrieve the functions relevant to it. Thus, the description of each function is treated as one document in the information retrieval process. To acquire the description of each function, we extract the name of the function, the names of the parameters of the function, the names of the local

variables in the function, and the names of all the directly invoked functions in the function.

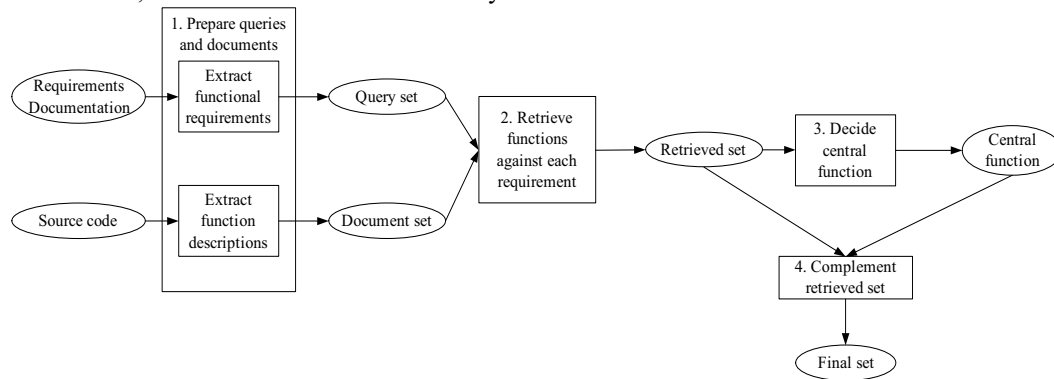


Fig. 1 The process of the approach

In the complementation part, we first identify the most important function (the central function) in the retrieved set to its associated requirement. Then we add functions on some paths related to the functions in the retrieved set into the final set. The call graph extracted from the source code will be used for finding these paths.

3.1. The Process

The process of our approach is depicted in Fig. 1. There are four main steps in our approach. The first step is to prepare queries and documents for the retrieving. As mentioned before, the main task of this step is to acquire the query set (the set of requirements) from the requirements documentation and the document set (the set of function descriptions) from identifiers extracted from the source code. The second step is to match each requirement against the function descriptions. Thus, for each requirement we can get a list of functions ranked by the similarity to the requirement. The third step is to identify the central function in the retrieved function set for each requirement. The calculation of the importance of each function is based on the similarity values acquired in the retrieval step. The fourth step is to further complement the retrieved function sets using the extracted call graph. For each retrieved function set, two kinds of functions will be added into the retrieved initial set. Firstly, the functions on the path from the entrance function to the central function and the decedent functions of the central function will be added. Secondly, the decedent functions of the functions that are non-central and not in the paths both from the entrance function to the central function and from the central function to its reachable decedent leaf nodes will also be added.

3.2. Preparing Queries and Documents

To apply the information retrieval technology, the basic step is to obtain both the query set and the document set. As our objective is to find the set of functions related to each requirement, we use the requirements documentation to form the query set, and the information of functions from the source code to form the document set.

The query set is formed as follows. From the requirements documentation, we pick out a paragraph of text for each atomic functional requirement. Usually, all the paragraphs are in a natural language (e.g. English). Then each paragraph is transformed into a set of index terms using the standard practice in information retrieval. That is to say, only the nouns and the verbs in the paragraph are considered in the transformation, and these words will be normalized to their original form (i.e. the single form of nouns and the infinitive form of verbs) to be the final index terms. Each set of index terms is a query in the query set.

The document set is acquired from the source code as follows. For each function in the source code, we extract the set of identifiers associated with the function. The identifiers include the name of the function, the names of the parameters of the function, the names of the local variables in the function, and the names of all the directly invoked functions in the function. As an identifier may not be in the standard form of a word, we preprocess the identifiers before we transform them into index terms. For example, an identifier in the form of several words connected by the symbol ‘_’ will be separated into several words. After the preprocessing, the words obtained from the identifiers will be transformed into a set of index terms using the same rules as mentioned above. Each set is a document in the document set.

3.3. Retrieving Initial Function Set

After both the query set and the document set are prepared, we use the vector space model for the retrieval. For each query in the query set, we will retrieve a subset of documents from the document set ranked by the similarity between the query and each document in the subset. We also set a threshold to control the size of each retrieved subset. According to the computation process of the vector space model described above, the retrieval process for one query in the query set can be described as follows. i) We use equation (2) to calculate each index term's normalized term frequency factor in certain vector which is used to express each query and document. ii) We calculate the general inverse document factor of each index term used for all vectors in the retrieval via equation (3). This value reflects the fact that the more frequently one index term occurs in all documents, the less important role the index term plays in the retrieval process. iii) We use the results of i) and ii) to calculate the weight of each index term in both the vector of the query by equation (5) and the vectors of all the documents by equation (4). iv) Finally, the similarity between a query and a document is calculated through equation (1). According to the degree of similarity between a query and a document, the documents against the query can be ranked in decreased order.

For all queries in the query set, we repeat the above process to acquire the initial function sets for all the requirements ordered by descending rank values.

3.4. Identifying the Central Function in a Retrieved Function Set

In each retrieved function set acquired in the second step, some related functions might not be included. It is due to the nature of the problem that some functions related to a requirement do not have any identifier representing the content of the requirement. In our approach, we try to use the call relationships between functions to catch those missed functions. The intrinsic difficulty of this catching is that some non-related functions might also be included. There are two main reasons for this difficulty. First, some non-related functions may be already included in the retrieved function set. Using these functions for the complementation is very likely to introduce more non-related functions. Second, some functions in the retrieved set are also related to some functions not related to the requirement. Thus, the complementation to these functions may also introduce those non-related functions.

To overcome this difficulty, we identify in the retrieved set a central function that is highly related and

not likely to be related to other requirements. The entire complementation process will mainly be based on this central function. For other non-central functions, we take a compromised method to complement them. The idea of the algorithm to decide the central function related to a requirement is much like the idea of the inverse document factor in IR technology. That is to say, the more requirements a function is related to, the less likely the function is to be a central function for any requirement.

Given a set of requirements, $rq = (rq_1, rq_2, \dots, rq_m)$, each of which has a set of retrieved functions, $f_i = (f_{i,1}, f_{i,2}, \dots, f_{i,n})$, related to it ($i=1, 2, \dots, m$). Each $f_{i,j}$ has a positive rank value, $r_{i,j}$, against the requirement rq_i . The algorithm for calculating all the central functions for all the requirements is depicted in Fig. 2. There are three main steps in the algorithm.

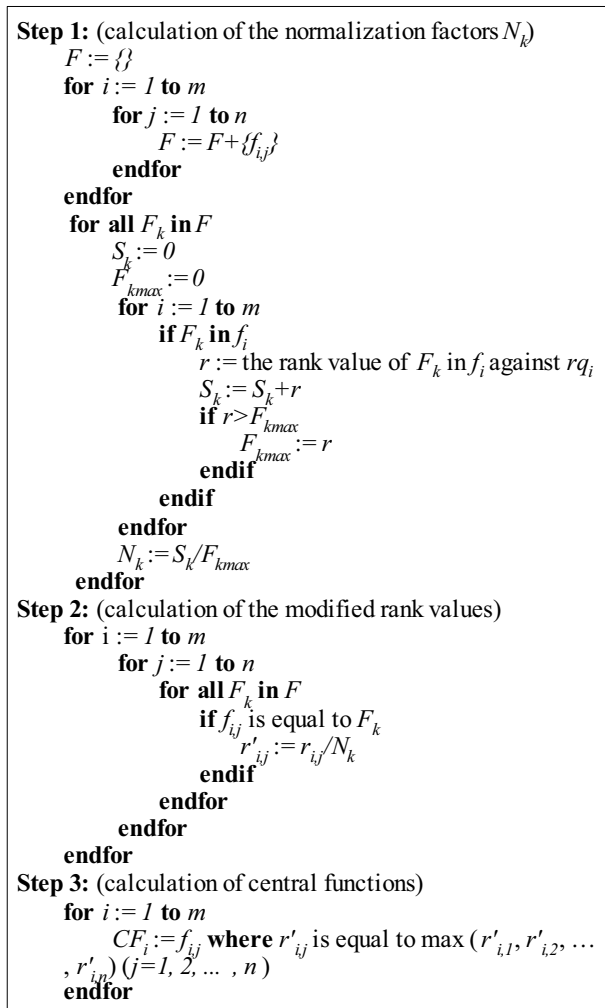


Fig. 2 Algorithm for identifying the central functions

i) We calculate a normalization factor for each function that appears in one or more retrieved sets. Firstly,

the set of all appearing functions, F , is calculated. Then, the sum of all the rank values (S_k) in all the retrieved sets and the maximum rank value (F_{kmax}) for each function F_k in set F are calculated. Finally, the normalization factor for F_k is defined as $N_k = S_k / F_{kmax}$. We do not use S_k directly as the normalization factor, because we want to ensure N_k to be a number greater than 1.

ii) We calculate the modified rank value ($r'_{i,j}$) for each function appearance in each retrieved set against each requirement. This value is defined as the original rank value divided by the function's normalization factor.

iii) Based on the modified rank values calculated in the second step, we select the function that has the highest modified rank value in each retrieved set as the central function for that retrieved set. For the i th requirement, the central function for the corresponding retrieved set f_i is denoted as CF_i in the algorithm.

3.5. Complementing the Retrieved Function Sets

After the central function is identified, we begin the complementation process. The basis of this process is obtaining the call graph from the source code. There have been standard methods for extracting a call graph, so we do not discuss it in this paper.

The process of the complementation is as follows. It takes the retrieved function set, the central function (which is a function in the retrieved function set) and the call graph as the input, and the complemented function set as the output. The main idea of this process is to complement the set of functions related both to the central function and the non-central functions in the call graph using different strategies. The final set is complemented based on the initial set. For the central function, we assume it is the only major function to implement a certain requirement. So we add all functions in the paths both from the entrance function to it and from it to its descendent leaf nodes to the final function set. For those non-central functions in the initial function set, we divide them into three groups: i) the functions that are in the paths from entrance function to the central function, ii) the functions in the paths from the central function to all its reachable leaf nodes, and iii) the functions that are not in above two groups of paths. For the first, we do nothing to them since they might be general functions that call the central function. Obviously, the complementation to the general function is very likely to introduce some functions implementing other requirements. For the second, we do nothing to them since all complementation for them has already been done. For the last, they should be functions that represent some extra implementation information of the requirement except the information the central function represents. We add the descendent functions for each such function. The relationships among the functions in the final set can be clearly observed in the call graph.

The process of the complementation takes three main steps:

i) Visit all paths that are from the entrance function to the central function, and add the functions in these paths to the final function set;

ii) Visit all paths that are from the central function to all the reachable leaf nodes in the call graph and add the functions in these paths to the final function set;

iii) For each of all the non-central functions in the initial set except those in the paths both from the entrance function to the central function and from the central function to its all reachable leaf nodes, visit all paths from it to its descendent leaf nodes and add the functions in these paths to the final function set.

An example of applying this algorithm is depicted in Fig. 3(a, b, c, d).

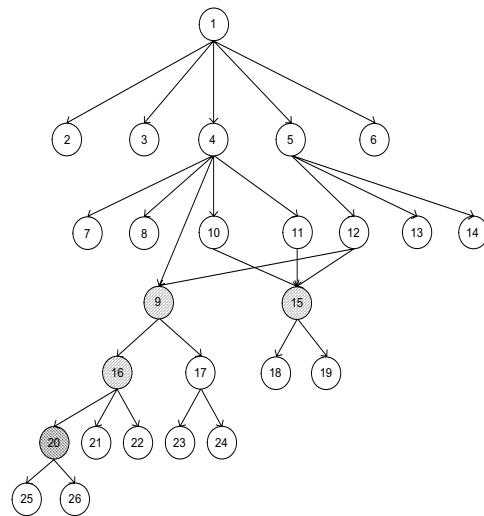


Fig. 3(a) Initially retrieved functions

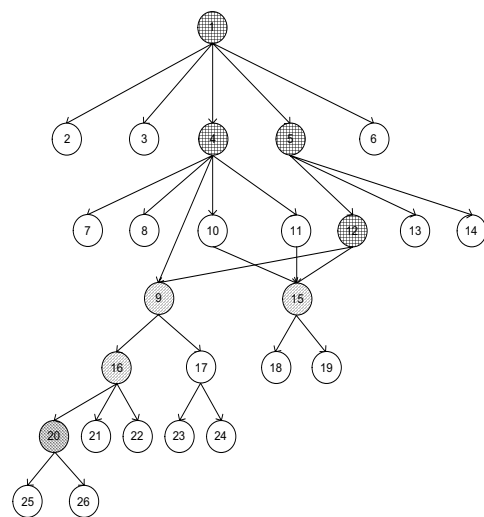


Fig. 3(b) Complementated set after step 1

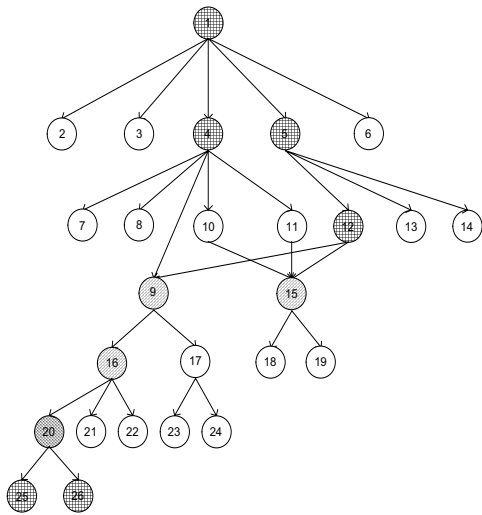


Fig. 3(c) Complemented set after step 2

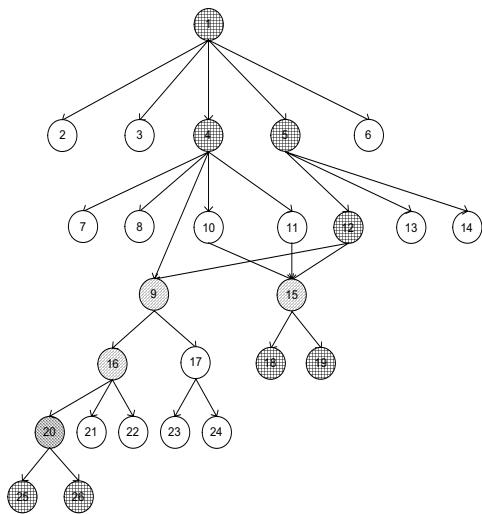


Fig. 3(d) Final function set after complementation

There are 26 functions for a program in the graph (labeled as 1, 2, ..., 26). The initially retrieved function set for a requirement is {9, 15, 16, 20} and the central function is function 20 as depicted in Fig. 3 (a). In the first step of complementation, we add function 1, 4, 5, and 12 into the final function set, which is shown in Fig. 3 (b). In the second step, we further add function 25 and 26 into the complemented set as depicted in Fig.3 (c). Then the non-central functions in the initial function set are function 9, 15 and 16. As function 9 and 16 are in the paths from the entrance function to the central function, we only consider function 15 in the next complementing step. In this final complementation step, we add function 18 and 19 into the final set, which is depicted in Fig. 3

(d). Therefore, the final function set is {1, 4, 5, 9, 12, 15, 16, 18, 19, 20, 25, 26}.

4. An Experimental Study

Our experimental study is based on an application system from GNU with the name DC [10] (which is distributed with the BC package). We use our approach to understand how this system's functional requirements are implemented in source code.

The DC system is a reverse-polish desk calculator which supports unlimited precision arithmetic, published by the Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA. The DC system has complete open source code and documentation that includes the detailed functional requirements expressed in a natural language (English). The DC system is about 2.7KLOC in ANSI C language, consisting 74 functions and 49 functional requirements.

4.1. Experimental Method

To apply our method for the analysis of this system, we use SMART [17] as the tool in the information retrieval step. SMART is an implementation of the vector space model of information retrieval proposed by Salton back in the 60's. The primary purpose of SMART is to provide a framework in which to conduct information retrieval research. Therefore, SMART can be viewed as the standard version of indexing, retrieval and evaluation for the vector space model.

For each requirement, based on the initial function set acquired from SMART, we pick out the central function and complement the set through the complementing strategy addressed in section 3.5 to decide the final function set for the requirement.

At the same time, in order to evaluate our approach, we also analyze this software system by ourselves to get the accurate function set related to each functional requirement. The comparison is then presented as the evaluation.

4.2 Experimental Results and Analysis

After the experiment, we have three groups of function sets. The first group is the accurate function sets, each of which is related to one functional requirement, which is acquired through understanding of the source code by experienced engineers. The second one is the initially retrieved function sets, each of which is related to one functional requirement retrieved by SMART. The third is the final function sets of our approach after the complementation. The first group is used as the relevant sets in the comparison. We compare the rest two groups

against the first one using two metrics, *precision* and *recall*, which are the common practice in the IR field, to evaluate our approach.

Precision is the ratio of the number of relevant functions retrieved for a given requirement over the total number of functions retrieved for that requirement. *Recall* is the ratio of the number of relevant functions retrieved over the total number of relevant functions. We use a cut level N to select the first N functions according to rank value in the retrieved initial function set and analyze the behavior of *precision* and *recall* with different values of N . For example, the relevant function set of a requirement is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} and the retrieved initial function set is {1, 2, 3, 4, 11, 12}. We use two cut levels (3 and 6) to calculate precision and recall. For the first cut, there are three functions in the retrieved initial function set, which are 1, 2, 3, and the relevant functions retrieved are also 1, 2, 3. Therefore, precision at cut one equals $3/3=100\%$, and recall equals $3/10=30\%$. For cut two, the retrieved initial function set is {1, 2, 3, 4, 11, 12}, and the relevant functions retrieved are 1, 2, 3, 4. Consequently, precision at cut two equals $4/6=66.7\%$, and recall equals $4/10=40\%$.

At the same time, we complement the retrieved initial function set at each different cut level and calculate the precision and recall for each complemented final function set.

We calculate the average values of precision and recall for all the requirements after we acquire their individual values of precision and recall for 5 cut levels (3, 6, 9, 12, 15). Fig. 4 shows the result.

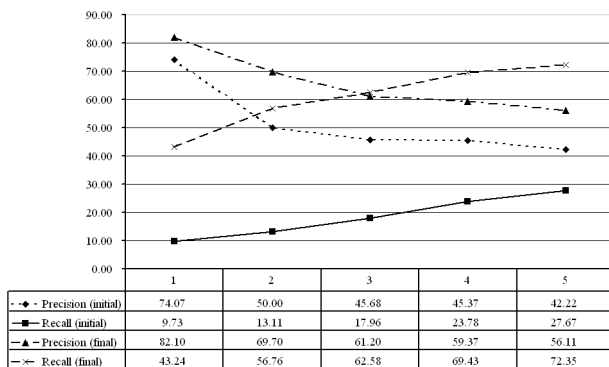


Fig. 4 The average result

For the retrieved initial set, the precision decreases as the cut level increases. The reason for this is that as the size of the retrieved set increases, the irrelevant functions introduced are more than the relevant ones. On the other hand, the recall increases as the cut level increases. This is because more relevant functions are added to the retrieved function set due to the enlargement of the function set. However, this value may not increase to 100 percent. The

explanation lies in that some functions of a requirement do not have the identifiers representing the content in the requirement. Therefore they cannot be retrieved in the retrieving step.

For the final set, which is acquired through complementing the initially retrieved set using call graph analysis, both the decreasing trend of precision and the increasing trend of recall are maintained. However, the actual values are both much larger than those of the initial set at the same cut level. The precision has an increase of 14.228 percentage points on average, and the recall 42.422. The prominent increase of the value of recall and precision demonstrates that the complementation of the function sets based on the retrieved result, is effective and precise. The complementation resolves to some extent the conflict between precision and recall in information retrieval technology. Generally, when recall reaches an acceptable value, the precision is hard to be acceptable. In our approach, we accomplish an acceptable tradeoff between these two metrics because of the effective complementation. As our complementing strategy is based on the premise that both the retrieved set and the central function we decide should be quite accurate, the overall goodness of the result also reflects the soundness of the retrieval step and the algorithm for deciding the central function.

For the best case of the experiment, the overall trends of the average result are maintained. The average increases between the final set and the initial set in precision and recall are 18.078 and 47.828 percentage points. For final set, the values at each cut level are larger than those in the average result. In particular, we can get about 70% for the value of precision and recall at the third cut level when we initially retrieve 9 functions for each requirement. It is shown in Fig. 5.

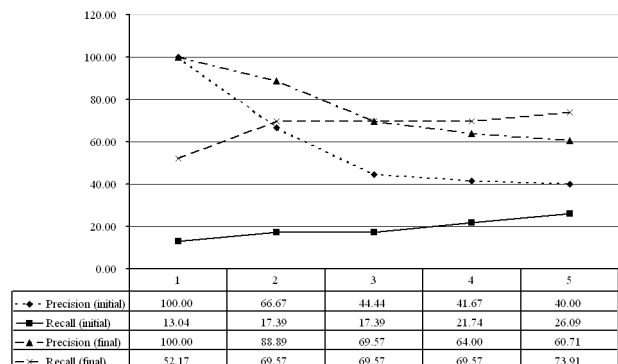


Fig. 5 The result on the best case

There are also some exceptional cases in the experiment, where the data of experiment behave unstably. Fig. 6 shows an exceptional example of a requirement.

In this unstable example, the value of precision neither achieves the highest value at the beginning of all cut levels nor decreases as the cut level increases. This result is caused by the imprecise result acquired in the retrieving step. The imprecision is introduced by some trivial description in the requirement, which can give less help to express the functional meaning of the requirement. This kind of trivial description happens to be matched with identifiers from some irrelevant functions, and thus the imprecision is introduced. However, as the relevant functions are retrieved with a high ratio contrast to the irrelevant, the value of the precision still presents an increasing trend as the number of retrieved functions increases. On this abnormal ranking result, we may get a wrong central function, and consequently, the complemented set may also be inaccurate and imprecise. In the worst case, our complementation strategy may result in a final set that is worse than its corresponding initial set in precision.

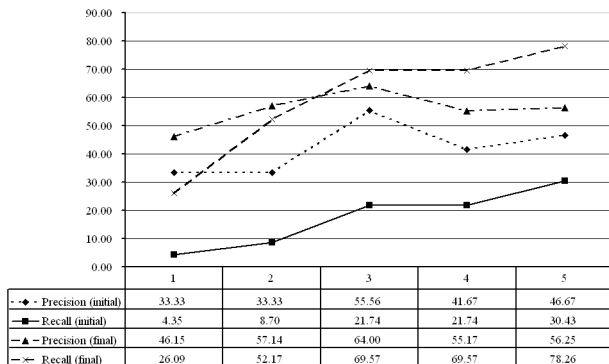


Fig. 6 The result on an unstable case

In Fig. 6, we get a similar result of complemented function set as the average result. In this case, it does not demonstrate the ineffectiveness of the complementing strategy. The reason is that the functions complemented for the wrong central function happen to be about the same as the functions that will be complemented for the actual central function.

4.3 Discussion

The approach we proposed in this paper to locate the relevant function set according to a certain requirement employs the IR model and the call graph. The partial matching of the vector space model avoids the disadvantage of most searching tools of editors which only retrieve exactly matched words or sentences. Despite the imprecision and incompleteness introduced with the partial matching at the meanwhile, we use some heuristics

to complement it. The approach has a good performance on the experimented DC system.

5. Related Work

5.1. Traceability between Source Code and Documentation

There are a variety of papers that address the traceability problem between source code and documentation. These papers can be classified into three main categories.

Firstly, some case tools, such as TOOR [13], IBIS [11], and REMAP [14], focus on maintaining the traceability between source code and documentation during the development process. Usually, a traceability database is used for recording the links between related artifacts.

Secondly, several methods are proposed for helping traceability recovery. The main idea of these approaches is to use some intermediate abstraction for both the source code and the documentation. Thus, the matching can be done on the basis of this abstraction. Usually, much manual work is need for transforming the source code and/or the documentation into this intermediate abstraction. For example, the abstraction in [18] is a hybrid visualization model based on static and dynamic information; the abstraction in [12] is a reflexion model; and that in [3] and [9] is an Abstract Object Language (AOL).

Thirdly, some researchers try to recover traceability using the IR technology (see e.g. [4], [2], and [1]) without any other abstraction for either source code or documentation. They use the identifiers from source code as queries and the texts from the documentation as documents. The primary goal of this approach is on mapping large entities (such as classes) in source code onto texts in documentation.

Our work is much deeper than the basic traceability problem. We are aiming at understanding how the requirements are implemented in source code. Therefore, although we also use the IR technology for retrieving the function set related to each requirement, our approach is fundamentally different from approaches for recovering traceability. Technically, our approach is aiming at fine-grained entities in source code and uses the call graph for further complementation.

5.2. Understanding System Behavior

Basically, the problem of understanding system behavior is to understand how a functional requirement is

implemented. There are both static and dynamic approaches for this problem.

In [6], Biggerstaff et al. view this problem as a concept assignment problem. They claim that the problem of understanding a program is to assign human-oriented concepts to program-oriented concepts. The assignment is based on a *priori* knowledge of the application domain. That is to say, the assignment is based on that people who try to understand the program have already known what the program do in general. Their approach always begins with a basic clue (such as suggestive data names, suggestive function names, or patterns of relationships), assign a human oriented concept to them as a starting point, and then propagate such behavior through the relationship among entities in source code (such as function calls, variable references, etc.). The domain knowledge that engineers have and the degree of understanding the program are accumulated interactively during the understanding process. Although their approach does help to understand a program in general, its basic idea of simulating the recognition process of human makes this kind of approaches difficult to be highly automatic.

Some papers that address the problem of the program behavior use the dynamic analysis or dynamic combining with the static analysis method. In [8], an approach is proposed to locate the functions related to a certain functional requirement. This method firstly gets the execution data by executing the program using the test cases created according to this functional requirement, and then uses the concept analysis technology to analyze the execution path information to locate the most related function to each functional requirement.

Our work addresses about the same problem but differs from both approaches. In our work, we acquire the domain knowledge from the functional requirements expressed in a natural language. Then we use the IR technology to accomplish the completely automatic assignment from human-oriented concepts to program-oriented concepts. Furthermore, static analysis is also adopted for the complementation. Our work is a static approach, which will not suffer from the limitations that a dynamic approach does and can be completely automatic.

6. Conclusion

In this paper, we have presented a static approach to locate the relevant functions according to functional requirements of a system. We use the IR method as the basis of our approach to retrieve the initial function set. The advantage of this step is that we can trace some implementation information acquired from the description of requirements to source code. Then, considering the incompleteness and imprecision of the IR method

exploited due to the nature of our problem, we complement this initial set through the analysis of the call graph extracted from source code. Our approach solves the conflict between precision and recall and can reach an acceptable tradeoff between them.

An experimental study is reported in this paper also. We make a complete and clear analysis about the experimental result. In the study, our approach works well on average. When the description of a requirement is focusing on the key actions it processes and does not have some trivial words, which may be mapped to irrelevant functions in source code, our approach can achieve its best result. Otherwise, our approach may include a negative effect when the requirement contains some trivial description and unfortunately this description has some mappings in source code. However, despite the possible negative effect, the effectiveness of our approach for this tested program is obvious. It should also be emphasized that our approach is completely static and can be automated by an assistant tool.

At this stage, we still cannot claim that we will get similar results on all the other systems although it might be quite expectable. In the future, we will focus on solving the imprecision introduced in the retrieving step and doing experiments on more software systems to evaluate our approach as well.

Acknowledgement

This effort is sponsored by the State 863 High-Tech Program (SN:2001AA113070), administered by the China National Software Engineering Research Institute.

References

- [1] G. Antoniol, G. Canfora, A. DeLucia, and E. Merlo, "Recovering Code to Documentation Links in Object-Oriented Systems," Proc. IEEE Working Conf. Reverse Eng., pp. 136-144, Oct. 1999.
- [2] G. Antoniol, G. Canfora, G. Casazza, A. DeLucia, and E. Merlo, "Tracing Object-Oriented Code into Functional Requirements," Proc. Eighth Int'l Workshop Program Comprehension, pp. 227-230, June 2000.
- [3] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-Code Traceability for Object Oriented Systems," The Annals of Software Eng., vol. 9, pp. 35-58, 2000.
- [4] G. Antoniol, G. Canfora, G. Casazza, A. DeLucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," IEEE Transactions on Software Engineering, vol. 28, no. 10, pp. 970-983, Oct.2002.
- [5] R. Baeza-Yates, B. Ribeiro-Neto, "Modern

- Information Retrieval,” pages 27-30, and 73-79, ACM Press, New York: Addison-Wesley, 1999, ISBN 0-201-39829-X.
- [6] T. Biggerstaff, B. Mitbender, and D. Webster, “The Concept Assignment Problem in Program Understanding,” Proc. Int’l Conf. Software Engineering, pp. 482-498, May 1993.
- [7] E. Chikofsky and J.C. II, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Software, vol. 7, no. 1, pp.13-17, Jan. 1990.
- [8] T. Eisenbarth, R. Koschke, D. Simon, “Locating Features in Source Code,” IEEE Transactions on Software Engineering, Vol. 29 Issue: 3, pp. 210-224 Mar. 2003.
- [9] R. Fiutem and G. Antoniol, “Identifying Design-Code Inconsistencies in Object-Oriented Software: A Case Study,” Proc. Int’l Conf. Software Maintenance, pp. 94-102, Nov. 1998.
- [10] GNU, “DC: An Arbitrary Precision Calculator,” (<http://www.gnu.org/directory/GNU/bc.html>)
- [11] J. Konclin and M. Bergen, “Gibis: A Hypertext Tool for Exploratory Policy Discussion,” ACM Trans. Office Information Systems, vol. 6, no. 4, pp. 303-331, Oct. 1988.
- [12] G.C. Murphy, D. Notkin, and K. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models,” Proc. Third ACM Symp. Foundations of Software Eng., 1995.
- [13] F.A.C. Pinhero and J.A. Goguen, “An Object-Oriented Tool for Tracing Requirements,” IEEE Software, vol. 13, no. 2, pp. 52-64, Mar. 1996.
- [14] B. Ramesh and V. Dhar, “Supporting Systems Development Using Knowledge Captured During Requirements Engineering,” IEEE Trans. Software Eng., vol. 9, no. 2, pp. 498-510, June 1992.
- [15] S. Rugaber and R. Clayton, “The Representation Problem in Reverse Engineering,” Proc. Working Conf. Reverse Eng., pp. 8-16, 1993.
- [16] G. Salton and M. E. Lesk, “Computer Evaluation of Indexing and Text Processing,” Journal of the ACM, 15(1):8-36, January 1968.
- [17] G. Salton, “The SMART Retrieval System - Experiments in Automatic Document Processing,” Prentice Hall Inc., Englewood Cliffs, NJ, 1971.
- [18] M. Sefika, A. Sane, and R.H. Campbell, “Monitoring Compliance of a Software System with Its High-Level Design Models,” Proc. Int’l Conf. Software Eng., pp. 387-396, 1996.