SNIAFL: Towards a Static Non-Interactive Approach to Feature Location

Wei Zhao, Lu Zhang, [1]Yin Liu, Jiasu Sun, Fuqing Yang
Software Engineering Institute, Peking University, Beijing, P. R. China, 100871
{zhaow, zhanglu, sjs, yang}@sei.pku.edu.cn
[1]mirainmay@hotmail.com

Abstract

To facilitate software maintenance and evolution, a helpful step is to locate features concerned in a particular maintenance task. In the literature, both dynamic and interactive approaches have been proposed for feature location. In this paper, we present a static and non-interactive method for achieving this objective. The main idea of our approach is to use the information retrieval (IR) technology to reveal the basic connections between features and computational units in source code. Due to the characteristics of the retrieved connections, we use a static representation of the source code named BRCG to further recover both the relevant and the specific computational units for each feature. Furthermore, we recover the relationships among the relevant units for each feature. A premise of our approach is that programmers should use meaningful names as identifiers. We perform an experimental study based on a GNU system to evaluate our approach. In the experimental study, we present the detailed quantitative experimental data and give the qualitative analytical results.

1. Introduction

During the past several decades, the heavy costs of maintaining existing software systems have become a great concern for many software projects. As estimated in [22], about 40 percent of the total cost of a software project is spent on software maintenance.

Usually, a maintenance task is to change or add some functionalities or features [27] [5], or refactoring the program without changing its behavior [12]. Although some refactoring tasks (such as the refactoring for generalization [21]) can be fulfilled automatically, most maintenance tasks require maintainers to spend more than half of their working time analyzing the documents and the source code to understand the features of the system being maintained [7]. A basic but very helpful step for this kind of maintenance is to locate interesting features in the source code [26].

More theoretically, the feature location problem can be formulated as identifying the relationships between the user's view and the programmer's view [23]. The user's view is made up of a collection of features denoted as $FEATURES=\{f_1, f_2, ...f_n\}$, while the programmer's view consists of a collection of computational units denoted as $UNITS=\{u_1, u_2, ...u_m\}$. Thus, the feature location problem is to recover the implementation relationships over $FEATURES \times UNITS$. In particular, two kinds of implementation relationships are usually distinguished [26] [11]. The first is the *relevant* relation, in which each feature is related to all the units contributing to the feature's implementation. The second is the *specific* relation, in which each feature is related only to the units that contribute to the feature's implementation but not to any other features' implementation.

There are mainly two categories of approaches addressing this problem. Firstly, interactive approaches based on maintainers browsing a graphical representation of the source code (such as [4], [6], [14] and [18]) can be used to assist maintainers to locate features. In the literature, this kind of approaches is also referred to as the static approaches. Secondly, automatic approaches based on dynamic execution of the system (see e.g. [24], [26] and [11]) are also reported in the literature. These approaches are usually referred to as the dynamic approaches.

In this paper, we propose a Static Non-Interactive Approach to Feature Location (SNIAFL). Like the dynamic approaches, our approach works in a batch-like manner without much human involvement. However, unlike the dynamic approaches, which use test cases to exhibit the basic relationships between features and units, we use information retrieval (IR) to achieve this objective. In fact, our approach is inspired by recent advances in applying IR for recovering traceability between code and documentation (see e.g. [2] and [16]). According to the characteristics of the retrieved results, we use the Branch-Reserving Call Graph [17] (an expansion of the call graph with branch information) to further recover the relevant and specific units and acquire the relationships among the relevant units for each feature.

2. Related Work

2.1. Feature Location

As mentioned above, the central task of feature location is to match the knowledge about features and that about computational units. In the previous research, two mainstreams of ideas for this task can be identified. The first one assumes that maintainers with the knowledge about features can browse through the source code of computational units to establish the connections. Therefore, the feature location problem is turned into building up an efficient support to facilitate maintainers for this browsing. This leads to the various interactive approaches. On the other hand, the second one assumes that maintainers can create test cases corresponding to features. As a result, the connections between features and computational units can be established via recording the execution traces of the test cases. Therefore, the feature location problem is turned into analyzing

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

execution traces with feature tags. This leads to the various dynamic approaches.

The forerunner of interactive feature location is [4], in which, this problem is referred to as a concept assignment problem. Thus, feature location is viewed as the process of assigning human-oriented concepts to program-oriented concepts. Several graphical representations of source code (such as the call graph, the program slice graph, and the program clustering graph) are exploited to facilitate this process. In [6], an interactive approach to feature location based on browsing the abstract system dependency graph (ASDG) is proposed. As ASDG can represent the dependencies among routines, types and variables at an abstract level, it can guide a user to search for the implementation of a particular feature. [14] reports the Aspect Browser, which can help maintainers to find feature implementations using lexical searches. This tool is based on Seesoft [9] and uses the map metaphor to graphically represent the location of the possible pieces of code for feature implementations. In [18], where features are referred to as concerns, the Concern Graphs is proposed as a facility of feature location. Compared to previous interactive approaches, the main difference is that the building of the Concerns Graphs is also interactive. Therefore, irrelevant source code will not be taken into consideration in the building process, and the piece of Concern Graphs used for locating a feature can be very small. As a result, this approach has a good scalability for large systems.

The main advantage of the interactive approaches is that the maintainer using such an approach can just have a vague idea of the target feature in the beginning and build up his or her knowledge in the process of feature location. However, the interactive nature makes these approaches very difficult to be highly automatic, and intensive human involvement is required.

The pioneer work of dynamic feature location is *Software Reconnaissance* [23][24]. In this approach, carefully designed test cases (among which, some are corresponding to a particular feature *f*, and others are not) are executed and the invoked computational units for each test case are recorded. Based on analyzing these units, four kinds of units regarding feature *f* can be distinguished: the *commonly involved* units, the *potentially involved* units, the *indispensably involved* units, and the *uniquely involved* units. A similar approach is reported in [26]. The main difference is that it can present code that is *unique* to a feature or *common* to a group of features at different granularity levels (i.e. files, functions blocks, lines of code etc.). Eisenbarth et al. have published several papers on using concept analysis for dynamic feature location (see [10] and [11] etc.). This approach uses a concept lattice to represent the execution traces recorded in the dynamic execution. Based on the lattice, several different relationships between features and computational units can be easily recovered.

The main advantage of the dynamic approaches is that they can automatically deal with many features in a batch-like manner after the test cases are acquired. However, they usually require a large number of test cases, and the design of these test cases may be a difficult task.

A simple empirical comparison of *Software Reconnaissance* and the approach in [6] is presented in [25]. The result of this comparison shows that *Software Reconnaissance* is more suitable for locating a number of features in a large but infrequently changed system, while the approach in [6] is more suitable for locating a specific feature under intensive changing.

2.2. IR-Based Traceability Recovery

In recent years, the use of information retrieval (IR) in recovering traceability between documentation and source code has become a focus. Antoniol et al. have published a series of papers on recovering code to documentation traceability (see i.e. [1] and [2]). In their approach, documentation pages are used as documents and summaries of *classes* in source code are used as queries. Two IR models (the probabilistic model [3] pp. 30-34 and the vector space model [3] pp. 27-30) are used in this approach without much difference in terms of performance. In [16], Marcus and Maletic use the Latent Semantic Indexing (LSI) method [8] [3] (pp. 44-46) (which is based on the vector space model) for recovering the documentation-to-code traceability. In this approach, source code files without any parsing are used as documents, and sections in the documentation are used as queries. According to the experimental results reported in [16], Marcus and Maletic's approach can to some extent outperform Antoniol et al.'s approach. Marcus and Maletic have also used the LSI method to define similarity measures between source code elements [15].

The traceability between documentation and source code recovered by the above two approaches is a kind of links between entities with large granularity, which are quite different from the entities (i.e. features and computational units) discussed in feature location. Therefore, these traceability recovery approaches are addressing a different problem other than the feature location problem. However, the use of IR does provide a means for connecting human-oriented knowledge and program-oriented knowledge. This is the starting point of our approach.

3. Approach Overview

In this section, we briefly present the objective of our approach and the main idea behind this approach. A similar application of this idea can also be found in our previous work [28].

As our approach is evaluated on a system written in the C language, we use the term *function* instead of *computational unit* for presenting our approach. In this paper, we concentrate on locating the *relevant* functions and the *specific* functions of a feature, although other relationships between features and functions can also be acquired via a slight extension of our approach. In this paper, the *specific* functions of a feature are defined as those functions that are definitely used to implement this feature but will not be used by any other features. The *relevant* functions of a feature are defined as all the functions that are involved in the implementation of the feature. Obviously, the *specific* function set is a subset of the *relevant* function set for every feature.

The goal of our approach is to solve the feature location problem statically. To achieve this goal, the basic idea is to use IR as the means to reveal the connections between features and functions, as indicated by recent studies on the effectiveness of using IR to recover traceability links. Obviously, the potential advantage of this idea is that it can save the costs for creating and executing test cases in dynamic approaches, and that for human involvement in interactive approaches. Like approaches to traceability recovery [2] [16], our idea requires that features

should be described in natural languages and meaningful identifier names should be used in the source code.

Due to the fuzzy matching nature of the IR technology, we cannot always acquire an accurate set of relevant functions for every feature directly from IR. This means that some irrelevant functions may be included due to the trivial description while some relevant functions may be excluded due to no corresponding descriptions in the features whatever IR method is used. As a result, we have to aim at correctly retrieving some specific functions for each feature when using IR since the specific description of each feature should not be lost. To achieve this, we use an IR model to decrease the importance of common words related to common functions, and for each feature acquire a function list ranked by the extent to which the function is specific to the feature. Then, an algorithm is involved to set a division point in the list. All functions before this division point will be used as the initial specific functions to each feature. We call them initial specific functions because some supporting specific functions, which are not mentioned in the feature descriptions, cannot be revealed through IR, and also they might not be completely correct.

After the initial specific functions for each feature are acquired, the next step is to acquire all the relevant functions for the feature. Besides, as no information about the call-relationships between the retrieved functions can be acquired from IR, it is also helpful to recover all the calling information. In our approach, we use a static representation of the source code for both purposes. The representation used in our approach is the Branch-Reserving Call Graph (BRCG), an expansion of the call graph with branching and sequential information, which is originally proposed for discovering use cases in source code [17]. This representation will be used to recover these relationships, and acquire those relevant functions according to the retrieved initial specific functions. Compared to the traditional call graph, the branch information in this representation can be used for eliminating irrelevant functions and refining call-relationships. Using this information, we can construct the pseudo execution traces for each feature. After all the relevant functions for each feature are determined, we can determine all the other relationships between features and functions, including the specific function sets that we are mostly interested in.

4. The SNIAFL Approach

4.1. The Process

The process of SNIAFL approach is depicted in Fig. 1. There are four main steps in the approach. The first step is to acquire the initial specific connections between features and functions. In this step, we use IR to filter the specific information of the features and recover these initial connections. The second step is to rank functions against each feature according to the retrieved result and choose the initial specific functions for each feature. The third step is to acquire the relevant functions and the possible pseudo execution traces using the BRCG extracted from source code. Based on each feature's initial specific functions, we complement all functions in the paths including the initial specific functions to acquire the relevant functions of the feature. As the BRCG maintains the branching and sequential information of source code, the possible pseudo execution traces of these relevant functions can also be acquired in the meantime. In the last step, we analyze the relevant functions to determine the final specific functions according to the definition of specific functions.

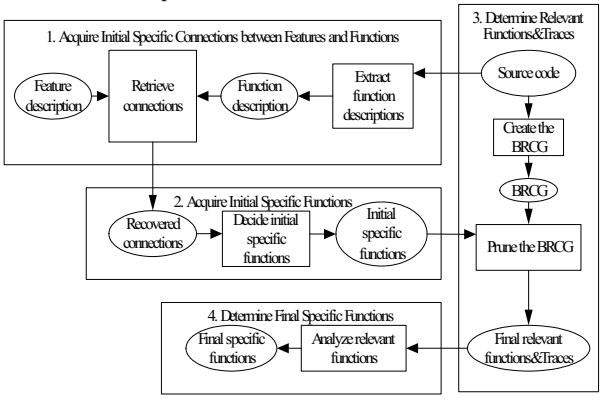

Fig. 1 The Process of SNIAFL Approach

4.2. Acquiring Initial Specific Connections between Features and Functions

4.2.1. The Vector Space Model

In our approach, we use the vector space model for indexing documents and queries and ranking the results. We introduce the vector space model in brief here. Please refer to [3] (pp. 27-30) for details.

The vector space model [19], [20] 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 $(w_{1,q}, w_{2,q}, \ldots, w_{t,q})$ represents the query q, in which $w_{i,q}$ is the weight of the *ith* index term in the query q, and t is the number of the index terms. Vector $(w_{1,j}, w_{2,j}, \ldots, w_{t,j})$ represents the document d_j, in which $w_{i,j}$ is the weight of the *ith* 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).

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

(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).

$$f_{i,j} = \frac{freq_{i,j}}{\max_l freq_{l,j}}$$

(2)

In equation (2), $freq_{i,j}$ is the raw frequency of the *ith* index term in the document d_j (i.e., the number of times the *ith* 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 *ith* index term in document d_j.

The computation of inverse document frequency for the *ith* index term, idf_i, is given by

$$idf_i = \log \frac{N}{n_i} \quad (3),$$

where N is the total number of documents in the system and n_i is the number of documents in which the *ith* 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 [3] (pp. 27-30), the two following equations can be used to compute the weights of index terms in documents and queries.

$$w_{i,j} = f_{i,j} \times idf_i \quad (4)$$

$$w_{i,q} = (0.5 + \frac{0.5\ freq_{i,q}}{\max_l freq_{l,q}}) \times idf_i \quad (5)$$

4.2.2. Preparing Queries and Documents

From the above introduction of the vector space model, we know that the nature of the vector space model is to treat the query and each document as the vectors and compute the similarity between them. *tf* and *idf* factors are used to measure the weights of index terms of the query and each document. The computations of *tf* and *idf* factors are based on the statistical data of all the documents. For the *tf* factor, the more appearances of one index term in a certain document, the higher the weight of this index term in this document. For the *idf* factor, the more appearances of one index term in all the documents, the less contribution the index term do to judge the similarity between the query and the documents.

As we are aiming at retrieving some specific functions for each feature, we can apply this *idf* factor to filter the specific information of the feature descriptions if we treat the features as the documents at the IR step. Thus, the common descriptions in different features will do less contribution to compute the similarity between the features and the functions, and the ranking will be mainly based on the specific descriptions in the features. Therefore, we treat the feature description set as documents, and the function description set as queries in our approach.

The set of feature descriptions (document set) can be acquired from the requirements documentation or domain experts or even users very familiar with the target system. For each feature, we will get a paragraph of text as its description. Usually, all the descriptions are in a natural language (e.g. English). Then each feature description is transformed into a set of index terms using the standard practice in IR. That is to say, only the nouns and the verbs in the description 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.

The function description set (query 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. As we are aiming at retrieving specific connections between features and functions, we do not want to incorporate those less specific identifiers from the body of 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 '_', or in the form of several words with capitalized first letters directly linked together, will be separated into several words. That is to say, both *feature_location* and *FeatureLocation* will be turned into *feature location*. 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 query in the query set.

4.2.3. Retrieving Initial Connections

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. Therefore we recover all the connections between the function and all the features. If there is no connection between a function and a feature, the rank value will be zero.

After we have done the above for all the queries in the query set, for each function we will have a list of features with similarity values. Then we can acquire a list of functions ranked by the similarity values for each feature through reorganizing the retrieval result. For example, there are n features in the feature set $F=\{f_1, f_2, ...f_n\}$ and m functions in the function set $U=\{u_1, u_2, ...u_m\}$. The similarity value between f_i and u_j is S_{ij} $(1 \le i \le n, 1 \le j \le m)$. The original retrieval result for function u_j is $\{f_1, f_2, ...f_n\}$ ranked by $S_{1j}, S_{2j}, ...S_{nj}$. The reorganized result for feature f_i is then $\{u_1, u_2, ...u_m\}$ ranked by $S_{i1}, S_{i2}, ...S_{im}$.

4.3. Identifying Initial Specific Functions

After acquiring the initial connections between features and functions considering the specific descriptions in the features, we identify the initial specific functions for each feature. In this step, for each feature, we sort the list of functions that have a connection (where the rank value is larger than zero) with it in descending order. We compute the distances between two consecutive functions for the function list. We simply use the arithmetic differences of the rank values of the functions as the distances between them. We use the position where the biggest distance appears as our division point to identify the initial specific functions. That is to say, the functions before this point will be chosen as the initial specific functions, while others not. It is obvious that, the functions before this point have much closer distances and therefore are more possible to have the same nature (i.e. the specific nature here) with the feature. The algorithm to determine the division point and choose the initial specific functions is depicted in Fig. 2.

Input: *u, m* (where *u* is the functions array with the descending order, *m* is the number of this array)
Output: *S* (the specific function set)
Step 1: $S \leftarrow \varnothing$
Step 2: **for** $i = 2$ **to** m
$d[i\text{-}1] \leftarrow$ absolute value of $(u[i].rankvalue\text{-}u[i\text{-}1].rankvalue)$
Step 3: $dmax \leftarrow d[1]$
$divisionpoint \leftarrow 1$
for $i = 2$ **to** m-1

if($d[i]>dmax$)
divisionpoint ← *i*
Step 4: **for** *i* = 1 **to** *divisionpoint*
$S \leftarrow S \cup \{u[i]\}$

Fig. 2 Algorithm to Determine the Division Point and Choose the Initial Specific Functions

The input is the descending function list for one feature with their rank values and the number of these functions. The output is the initial specific functions to the feature.

The first step is to initialize the output initial specific function set. Step 2 calculates all distances between two consecutive functions sorted in descending order. In the third step, for all acquired distances, the biggest distance is chosen and therefore the division point is determined. In the final step, all functions before the division point will be chosen as the initial specific functions in our approach. Obviously, the worst case time complexity of this algorithm is *O(m)*, where *m* is the number of functions.

4.4. Determining Relevant and Specific Functions

After the initial specific functions are identified, we begin to determine the relevant and specific functions. The basis of this step is obtaining the BRCG from the source code. We traverse the BRCG and determine the relevant functions according to the initial specific functions, and finally calculate the specific functions.

4.4.1. The BRCG

The Branch-Reserving Call Graph (BRCG) is firstly introduced in [17] for discovering use cases. This structure is a representation of the source code by including branch information into the traditional call graph. In this structure, both branch statements and function-call statements are considered. For the simplicity, the *if*-statements, the *case*-statements and all the loop statements are all treated as branch statements.

Each node in the BRCG is a function, a branch statement, a branch in a branch statement, or a return statement. Furthermore, the loop statement is simply regarded as a two-branch condition statement; one branch does not have any nodes, and the other has all the nodes belonging to the loop. The relationships between the nodes are the sequential relationships and the branching relationships. Therefore, the BRCG can be formally defined as follows: the BRCG is a triple $BRCG=<N, S, B>$, where:

i) N is the set of functions, branch statements, branches in branch statements, and return statements;

ii) S is the set of sequential relationships, where for $\forall <n_1, n_2> \in S$, $n_1 \in N$ and $n_2 \in N$;

iii) B is the set of branching relationships, where for $\forall <n_1, n_2> \in B$, $n_1 \in N$ and $n_2 \in N$; and

iv) for $\forall <n_1, n_2> \in S$ and $\forall n_3 \in N$, $<n_1, n_3> \notin B$, and for $\forall <n_1, n_2> \in B$ and $\forall n_3 \in N$, $<n_1, n_3> \notin S$.

The first three conditions define that the BRCG is a graph with two kinds of relationships. The fourth condition defines that a node connects to its sub-nodes only through one kind of relationships. An example of the BRCG is depicted in the following two figures. Fig. 3 depicts the source code of a function, and Fig. 4 depicts the corresponding BRCG.

The construction of the BRCG for the whole system is based on the sub-BRCGs of all the functions. The entire BRCG of a system can be acquired through connecting all the sub-BRCGs for the functions into an entire graph by linking the root node of each sub-BRCG with the nodes of the same signature in other sub-BRCGs. Please refer to [17] for the algorithm to build the BRCG for each function from source code. In our approach, we do not include the edges of recursive function calls in the BRCG.

```
void foo()
{       f1();
        if(condition)
        {       f2();
                return;
        }
        else
        {       while(condition)
                {       f3()
                        f4();
                }
                f5();
        }
        f6();
}
```

Fig. 3 Sample Code for BRCG

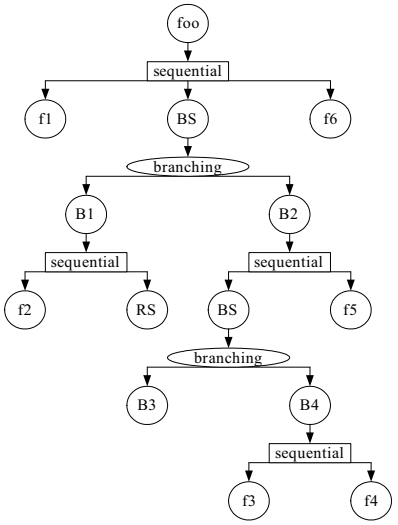

Fig. 4 A Sample BRCG

4.4.2. Acquiring Relevant Functions using BRCG

Since the specific functions of a feature mean that the implementation of this feature will definitely invoke the functions and the implementation of other features will not invoke them, it is very likely that the branches that will not invoke any of such functions are not relevant to the feature. Therefore, we can prune some branches according to the non-existence of the specific functions.

For example, if one branch statement BS_i has two branches $BS_i\_b_1$ and $BS_i\_b_2$, and $BS_i\_b_1$ includes some specific functions while $BS_i\_b_2$ not, the branch $BS_i\_b_2$ will be pruned according to the definition of specific functions. Furthermore, if the branch statement BS_i lies in one of the branches ($BS_j\_b_1$) of another branch statement BS_j, the specific nature of the $BS_i\_b_1$ can be propagate to $BS_j\_b_1$. And the propagation ends at the root node of the BRCG. This is the main idea we use to prune BRCG. Obviously, all the functions left in the pruned BRCG should be the relevant functions to the feature. The algorithm for pruning the BRCG is depicted in Fig. 5.

The input of this algorithm is the BRCG with root node *n*, the set of functions *F* that includes all the appearances of the

initial specific functions and the number of this set *m*. The output is the pruned BRCG.

Input: a BRCG with the root node *n*, the existences set *F* of the initial specific functions and the number of the set *m*.
Output: the pruned BRCG.
Step 1:
for each existence f_i in set *F* (*i* = 1, 2, ..., *m*)
begin
currentnode = f_i
while (*currentnode* ≠ *n*)
begin
if *currentnode*'s parent node is a branch
begin
create a specific relation between this branch and its branch statement
currentnode=*currentnode*'s parent node's parent node
end
else if *currentnode*'s parent node is a function
currentnode=*currentnode*'s parent node
end
end
Step 2: traverse the BRCG marked by specific relation from *n*
currentnode=*n*
if *currentnode* is a leaf node
stop
if *currentnode* is a function
traverse all its sub nodes as step 2
if *currentnode* is a branch statement
begin
if there is at least one specific branch of it
prune all the non-specific branches from the BRCG
traverse all sub nodes of all the rest branches as step 2
end
Step 3:return *n*

Fig. 5 Algorithm to Prune the BRCG According to the Initial Specific Functions

This algorithm includes two main steps. The first step marks all branches where the initial specific functions are. The process of propagating the specific branches ends at the root node of the BRCG. The second step prunes those unmarked branches that have the same branch statement with those marked branches. The definitely used nodes and their sub-nodes will always be maintained in the pruned BRCG. This step is recursive, and it traverses the BRCG from root node with depth first strategy. When the node is a branch statement and there is at least one specific branch of it, all the non-specific branches will be deleted. Obviously, if there is more than one specific branch in a branch statement, all of them will be maintained in the BRCG. All functions existing in the pruned BRCG are the relevant functions of the feature. The worst-case time complexity of this algorithm is linear to the number of nodes in BRCG, which is then proportional to the size of the source code.

4.4.3. Determining Specific Functions

After we acquire all the relevant functions of each feature from the pruned BRCG, we can determine all the other relationships between features and functions in the same way a dynamic approach deals with the recorded functions invoked for each feature. Due to the space limit, we only present the algorithm for calculating the final specific functions for each feature, which is depicted in Fig. 6. This algorithm is based on the relevant relationships between the features and the functions. We construct a two-dimension array to store this relation. The rows denote all the features and the columns denote all the functions. In the algorithm, *V* is the two-dimension array. *V*[*i, j*] equals 1 if the *j*th function is the relevant function to the *i*th feature, otherwise 0.

Input: *V*, *n*, *m* (where *V* is the relevant array, *n* and *m* are the number of rows and columns respectively),
Output: *S* (the specific array)
for *i* = 1 to *n*
for *j* = 1 to *m*
if (*V*[*i, j*])
begin
isSpecific←true
for *k* = 1 to *n*
if (*V*[*k, j*] and *k*≠*i*)
isSpecific←false
if (*isSpecific*)
S[*i, j*] = 1
else
S[*i, j*] = 0
end

Fig.6 Algorithm to Determine the Specific Functions

The idea of this algorithm is to check each feature like this: 1) It finds the first function that is relevant to this feature. 2) It checks whether this function is also relevant to another feature. 3) If no, it marks the function to the feature with 1 in the output array, and otherwise 0. 4) It finds another relevant function, and does the same. 5) After all the functions are processed, the specific functions for one feature can be acquired. The worst case time complexity is $O(n^2m)$.

4.5. Recovering Relationships between Functions

As the BRCG includes branching and sequential information, we can generate each possible execution trace by traversing the graph. For the relevant functions acquired in the pruned BRCG, we use the following algorithm to generate the possible execution traces. We call these traces pseudo execution traces since we do not acquire these traces by real execution and there are some simplifications in the BRCG (such as simplifying loop statements as branch statements simply). Fig.7 shows the algorithm.

Input: a BRCG, a root node *n* in the BRCG
Output: a set of traces
Step 1:
if *n* is a leaf node and not a return statement
return{"enter_"+*n*.label+ "-"+"exit_"+*n*.label}
if *n* is a return statement
return{"return"}
Step 2: *S*=∅
Step 3: **for** each n_i as a sub-node of *n*
generate the traces of n_i into S_i
Step 4: **if** the relationships between *n* and its sub-nodes are sequential
begin
if *n* represents a function
U={"enter_"+*n*.label+ "-"}
else
U={""}
for each S_i

```
        begin
            T:= ∅
            for each trace t in S_i
                for each s in U and s does not end with
"return"
                    T:= T∪{s+t}
            U:=T
        end
    if n represents a function
        for each s in U
            S:=S∪{s+"-exit_"+n.label}
    else
        S:=U
end
Step 5: if the relationships between n and its sub-nodes are
branching
    for each S_i
        S:=S ∪ S_i
Step 6: return S
```

Fig. 7 Algorithm to Generate Pseudo Execution Traces

This algorithm is a recursive algorithm. During the generation, we only generate trace information for non-leaf nodes that represent functions and all the leaf nodes.

5. An Experimental Study

In our experimentation we used a software system of GNU, named DC [13] (which is distributed with the BC package). We acquired the complete source code and the requirements specification documentation of the DC system, which contains 49 functional requirements and 74 functions. Among the 49 features, there are 21 primitive features, each of which implements a single functionality of the system. The others are composite features. Their location can be acquired through the analysis of all the locations of their component features.

For example, a primitive requirement which implements the binary operation "add" is described in DC's requirements specification as "Pops two values off the stack, adds them, and pushes the result". Several representative relevant functions of this requirement are "dc_push, dc_binop, dc_pop, dc_add", and the specific function is "dc_add".

5.1. Experimental Method

To apply our method for the analysis of this system, we used SMART [20] as the tool in the IR step. SMART is an implementation of the vector space model of IR proposed by Salton back in the 60's. The primary purpose of SMART is to provide a framework to conduct IR research. Therefore, SMART can be viewed as the standard version of indexing, retrieval and evaluation for the vector space model.

For all the 21 features, we applied SNIAFL to get the relevant functions, the possible pseudo execution traces and the specific functions. For each feature, we got three groups of data by our approach: one group is the initial specific functions and the final specific functions of each feature; the second group is the relevant functions acquired by the BRCG using the initial specific functions; and third are the pseudo execution traces of these relevant functions constructed by the BRCG.

To evaluate SNIAFL, we manually analyzed the DC system, and for each of the 21 features, we recorded the genuine relevant functions, the genuine execution traces for them and the genuine specific functions. As a comparison, we also designed test cases for each feature to execute an instrumented version of the DC system to get the dynamic results. As dynamic approaches are heavily dependent on the quality of the test cases, there will not be complete and/or precise results if the test cases are not sufficient or well designed. Therefore, for each feature, we designed two groups of test cases to imitate the test cases by an experienced maintainer and those by a common maintaining engineer. Each test case exactly invoked one execution trace, the functions invoked by the test cases for a feature were recorded as its relevant functions, and all the relevant function sets were used to calculate the specific functions for each feature in a way similar to the algorithm depicted in Fig. 6. To confirm the necessity of using the function to feature retrieving strategy and the BRCG in our approach, we also recorded the retrieval results of using features as queries and functions as documents as the relevant functions. We used 0.1 as the threshold to choose functions with higher similarity as the relevant functions.

Therefore, SNIAFL was evaluated from three aspects: the relevant functions, the execution traces, and the specific functions. For the relevant functions, we compared the results of SNIAFL with the results of the dynamic approach, the results retrieved directly from the SMART using features as queries and functions as documents, and the genuine results. We used *precision* and *recall* to do the comparison. *Precision* is the ratio of the number of correct functions acquired for a given feature over the total number of functions acquired for that feature. *Recall* is the ratio of the number of correct functions acquired over the total number of accurate relevant functions. For the execution traces, we compared the results of SNIAFL with those of the dynamic approach and the genuine results. For the specific functions, we compared the initial results, the final results and the results of the dynamic approach with the genuine results.

5.2. Results on Acquiring Relevant Functions

5.2.1. Quantitative Results

We calculate the *precision* and *recall* of the relevant functions acquired by the three approaches. Table 1 shows the results of the three approaches for each feature. Since the functions invoked by the test cases for any feature should be relevant to the feature, the *precision* of the dynamic approach was always 100 percent. Therefore, we do not list it in the table.

Table 1. The Recall and Precision of Relevant Functions of Three Approaches

No.	IR Only		Recall of Dynamic Approach		Our Approach	
	Recall	Precision	Insufficient	Well-designed	Recall	Precision
1	12.50%	14.29%	100%	100%	100%	100%
2	14.29%	33.33%	100%	100%	100%	100%
3	6.67%	22.22%	63.33%	90 %	96.67%	96.67%
4	5.71%	22.22%	68.57%	91.43%	100%	94.59%
5	14.29%	31.25%	71.43%	91.43%	100%	94.59%

6	8.57%	60 %	71.43%	91.43%	100%	94.59%
7	17.14%	37.5%	71.43%	91.43%	100%	94.59%
8	11.43%	33.33%	71.43%	91.43%	100%	94.59%
9	8.57%	42.86%	71.43%	91.43%	94.29%	89.19%
10	11.43%	36.36%	71.43%	91.43%	100%	94.59%
11	8.57%	30 %	71.43%	91.43%	100%	94.59%
12	14.29%	35.71%	71.43%	91.43%	100%	94.59%
13	11.76%	36.36%	70.59%	91.18%	100%	94.44%
14	3.03%	14.29%	66.67%	90.91%	100%	94.29%
15	15.15%	38.46%	72.73%	90.91%	100%	84.62%
16	11.76%	36.36%	67.65%	91.18%	100%	89.47%
17	8.82%	27.27%	76.47%	91.18%	100%	94.44%
18	17.65%	46.15%	67.65%	91.18%	100%	89.47%
19	14.71%	38.46%	67.65%	91.18%	100%	89.47%
20	12.50%	26.67%	65.62%	90.63%	100%	94.12%
21	3.70%	20.00 %	62.96%	88.89%	100%	37.5%
Avg.	11.07%	32.53%	72.44%	91.91%	99.57%	90.97%

For the results of the IR method, neither precision nor recall is good enough. The average recall is 11.07%, the worst is 3.03%, and the best is only 17.65%. The average precision of the IR method is 32.53%, the worst is 14.29%, and the best is 60%.

The dynamic approach is much better, but the pre-condition is that the test cases are well designed. In our experiment, the results on the two groups of test cases can confirm this. The recall of the dynamic approach with the insufficient test cases is much lower than that with the well-designed test cases. On average, the recall of the insufficient test cases is 72.44%. The well-designed test cases have a higher recall (91.91%), but still cannot reach 100 percent. This is because some unusual error handling branches are not invoked by those test cases. Our approach can avoid this weakness.

For SNIAFL, recall is 99.57% and precision is 90.97% on average. The recall is higher than the dynamic approach and close to 100 percent. The average result confirms the effectiveness of SNIAFL to acquire the relevant functions to some extent. However, we still have an exceptionally bad case in which the precision is only 37.5%.

5.2.2. Qualitative Analysis

The bad performance of IR only approach is due to the imprecise nature of this technology. For the low recall, it is because some relevant functions of a feature do not have identifiers representing the content in the feature description. Therefore, they cannot be acquired by the IR only method. For the not high enough precision, the explanation lies in the feature to function retrieval strategy. Some commonly used words in the description of a feature will lead to retrieve all the functions having some of those words as identifiers. The dynamic approach is much more precise. Its only disadvantage is the dependence on the quality of test cases.

The good recall of SNIAFL lies in two factors. Firstly, the static nature of SNIAFL causes it to take more than necessary functions as relevant. Secondly, the function to feature retrieval strategy should be quite effective, and thus good initial specific functions are usually selected. However, the imprecision of the initial specific functions results in that we cannot reach the completely correct result. The precision of our approach is not so good as the recall. This is due to the conservative nature of the static approach that takes all possibility into consideration. Especially in the worst case the precision is down to 37.5%. In this case, the semantic information of the program does much effect to the implement of this feature. Our approach cannot get such information and therefore collects some functions in such branches that are syntactically relevant but semantically irrelevant.

5.3. Results on Acquiring Function Relationships

5.3.1. Quantitative Results

Due to the simplification of the loop statement, the pseudo execution traces acquired from BRCG are not exactly same with the traces from dynamic execution in some cases. In these cases in our experiment, we still treat the acquired traces as the correct one to be calculated.

Table 2 shows the execution traces acquired by SNIAFL, the dynamic approach and the genuine traces to each feature.

Table 2. Execution Traces

No	Dynamic Approach		Our Approach		Genuine
	Insufficient	Well-designed	Generated	Correct	
1	1	1	3	2	2
2	1	1	1	1	1
3	6	24	756	0	42
4	6	24	756	36	36
5	3	12	504	30	30
6	3	12	504	30	30
7	3	12	504	30	30
8	3	12	504	30	30
9	3	12	504	0	30
10	3	12	504	30	30
11	3	12	504	30	30
12	3	12	504	30	30
13	3	12	252	27	27
14	6	24	756	36	36
15	6	24	2106	42	42
16	6	24	1008	48	48
17	3	12	504	0	30
18	3	12	1008	18	18
19	3	12	1008	30	30
20	3	12	252	18	18
21	3	12	>10000	18	18

For the dynamic approach, each test case invokes exactly one execution trace. It is obvious that the quality of test cases determines the result. For the well-designed test cases, there are still some traces lost, while for the insufficient test cases, more traces are lost. Except three features, SNIAFL can acquire all the genuine traces. The main problem with this approach is that there are too many traces generated. This is due to the static nature of our approach.

5.3.2. Qualitative Analysis

To be honest, our approach is not quite effective in acquiring execution traces for each feature. However, our approach can reveal some unusual traces for most features. This might be helpful in some special cases. To reduce the traces generated by our approach, we need to apply more restrictive static analysis methods to get rid of some wrongly generated traces.

5.4. Results on Acquiring Specific Functions

5.4.1. Quantitative Results

Table 3 shows the results of initial and final specific functions of our approach and the results of the dynamic approach with insufficient and well-designed test cases in the experimentation. Due to the space limit, we use a concise way for presenting the results.

Table 3. Specific Functions

	Totally correct	Totally wrong	Partially wrong	Correct ratio
Initial	14	3	4	66.67%
Final	18	3	0	85.71%
Insufficient	20	0	1	95.24%
Well-designed	21	0	0	100%

Of all the 21 features in our experimentation, 66.67% of features acquire the completely correct specific functions during the IR step. Despite the imprecise nature of IR, the experimental results show that our approach is effective to some extent. Furthermore, we acquire the 85.71% correct ratio of the specific functions after analyzing the relevant functions. This ratio is quite acceptable practically.

For dynamic approach, the correct ratio of the insufficient test cases is 95.24%, while the well-designed test cases 100%.

5.4.2. Qualitative Analysis

Due to the imprecise nature of IR technology, the effectiveness of the initial specific functions reflects the effectiveness of our retrieval strategy. We use features as documents and functions as queries to avoid commonly used functions being very similar to any feature. The algorithm for selecting initial specific functions will allow only very similar functions to be selected.

For the final specific functions, we analyze the relevant functions according to the definition of specific functions. This step is only effective to those partially wrong initial specific functions. The explanation is that we can eliminate the non-specific functions from the partially wrong initial specific functions and complement those supporting ones, and for those totally wrong, we can do nothing at this stage.

For the insufficient test cases of the dynamic approach, some non-specific functions are picked out due to the insufficient execution traces.

5.5. Threats to Validity

The main threat to validity is to what extent the experimented system is representative of all the possible target systems in practice. Although DC is a real world system for various versions of UNIX and Linux, its size is still small compared to a typical system in practice. This threat can be reduced via experimenting on more and larger systems. Another threat is the test cases used for dynamic feature location in the comparison. As we are not professional testers, we cannot ensure that the well-designed test cases are still so in the eyes of professional testers. However, this threat may not be very effective in the experiment because DC is quite a simple system. Therefore, it is quite easy for non-professional testers to create well-designed test cases.

6. Discussion

6.1. Automatic vs. Interactive

Compared to previous static approaches to feature location, a distinct characteristic of our approach is the ease of automation. This is achieved by conceding the following price. A maintainer using our approach has to describe and retrieve all the features before locating a specific feature. This may cause some inconvenience in practice. Furthermore, the non-interactive way of our approach will prevent a maintainer from building up the knowledge about the feature in the locating process. However, we think the gaining from the automatic nature can offset the price by saving much human involvement.

The static representation used in our approach is also very simple compared to other representations used in static approaches. However, this representation is suitable for computer processing and there is no much imprecision incorporated when abstracting it from the source code.

6.2. IR vs. Test Cases

Compared to dynamic approaches, a clear difference of our approach is the use of IR instead of test cases as the driving force. The advantage of using test cases is that the more test cases are involved, the more precise the result is, and there are no false positive relevant functions if no test case is wrongly credited to a feature. In our approach, we have to concede the imprecise nature of IR, and therefore incorporate errors in the first place. However, our approach can save the cost of designing and executing so many test cases.

6.3. Granularity

The main weakness of our approach is the lack of flexibility of choosing granularity. The way of using IR to set up connections between features and functions determines that the granularity has to be at the function level. There are circumstances that feature implementations should be represented at the level of fragments of functions or even the statement level. As we cannot abstract specific descriptions for entities smaller than functions, it is difficult for our approach to support finer granularity. For many of other approaches, there is much more flexibility for choosing functions, branches or statements as the basic computational units.

7. Conclusions and Future Work

In this paper, we have proposed a static and non-interactive approach to locating relevant and specific functions for all the features. The starting point of our approach is to locate some initial specific functions to each of features through IR. Based on the initial specific functions, we recover all relevant functions through navigating a static representation of the code named BRCG using some algorithms. Due to the characteristics of the BRCG, we also acquire the pseudo execution traces for each feature.

An experimental study is reported in this paper also. We evaluate our approach from three aspects respectively – the relevant functions, the pseudo execution traces, and the specific functions. In the experiment, our approach works well on average, especially for specific functions and the relevant functions. For the recovered pseudo execution traces, there are too much irrelevant traces generated by our approach. However, our approach does find some unusual traces. In general, despite of this weakness, the overall effectiveness of our approach for this tested program is obvious.

In the future, we will focus on doing experiments on more software systems to further evaluate the feasibility and usability of our approach. To apply IR more effectively, we will exploit more sophisticated preprocessing mechanisms to deal with identifiers using abbreviations and/or acronyms. We will also look at possible ways for reducing the number of pseudo execution traces generated by our approach.

Acknowledgements

This effort is sponsored by the National 973 Key Basic Research and Development Program No. 2002CB31200003, the State 863 High-Tech Program No. 2001AA113070, and the National Science Foundation of China No. 60125206, 60233010 and 60373003.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. DeLucia, "Information Retrieval Models for Recovering Traceability Links between Code and Documentation," Proc. IEEE International Conf. Software Maintenance, pp. 40–49, Oct. 2000.

[2] G. Antoniol, G. Canfora, G. Casazza, A. DeLucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation," IEEE Transactions on Software Engineering, 28(10), pp. 970-983, Oct.2002.

[3] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, ACM Press, New York: Addison-Wesley, 1999, ISBN 0-201-39829-X.

[4] T. Biggerstaff, B. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," Proc. International Conf. Software Engineering, pp. 482-498, May 1993.

[5] S. Bohner and R. Arnold, "An Introduction to Software Change Impact Analysis", Software Change Impact Analysis, IEEE Computer Society, 1996.

[6] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," Proc. International Workshop Program Comprehension, pp. 241-249, June 2000.

[7] T. A. Corbi, "Program Understanding: Challenge for the 1990's," IBM Systems Journal, 28(2), pp. 294-306, 1989.

[8] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," Journal of the American Society for Information Science, 41, pp. 391-407, 1990.

[9] S. Eick, J. Steffen, and E. Summer, "Seesoft—A Tool for Visualizing Line-Oriented Software Statistics," IEEE Transactions on Software Engineering, 18(11), pp. 957-968, Nov. 1992.

[10] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis," Proc. International Conf. Software Maintenance, pp. 602-611, Nov. 2001.

[11] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," IEEE Transactions on Software Engineering, 29(3), pp 210-224, March, 2003.

[12] M. Fowler, Refactoring - Improving the Design of Existing Code. Addison-Wesley, 1999.

[13] GNU, "DC: An Arbitrary Precision Calculator," (http://www.gnu.org/directory/GNU/bc.html)

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

[15] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," Proc. International Conference on Software Engineering (ICSE 2001), pp. 103-112, May, 2001,.

[16] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," Proc. International Conf. on Software Engineering, pp. 125 –135, May 2003.

[17] T. Qin, L. Zhang, Z. Zhou, D. Hao, and J. Sun, "Discovering Use Cases from Source Code using the Branch-Reserving Call Graph," Proc. Asia-Pacific Software Engineering Conference, pp. 60-67, December 2003.

[18] M. P. Robillard and G. C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Depnedencies," Proc. International Conference on Software Engineering, 3-10 May 2002, pp. 406 –416.

[19] G. Salton and M. E. Lesk, "Computer Evaluation of Indexing and Text Processing," Journal of the ACM, 15(1):8-36, January 1968.

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

[21] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for generalization using type constraints," Proc. Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 13-25 , October 2003.

[22] R. J. Turver and M. Malcolm, "An Early Impact Analysis Technique for Software Maintenance," Journal of Software Maintenance: Research and Practice, 6(1), pp.35-52, January-February 1994.

[23] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg, "Locating User Functionality in Old Code," Proc. International Conference on Software Maintenance, pp. 200-205, Nov. 1992.

[24] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," J. Software Maintenance: Research and Practice. Vol. 7(1), pp. 49-62, Jan. 1995.

[25] 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(2), pp. 105-114, 2003.

[26] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating Program Features using Execution Slices," Proc. Symposium on Application-Specific Systems and Software Engineering Technology, pp. 194-

203, March, 1999.

[27] S.S. Yau, R.A. Nichol, J.J. Tsai and S. Liu, "An Integrated Life-Cycle Model for Software Maintenance", IEEE Transactions on Software Engineering, 15(7), pp 58-95, July 1988.

[28] W. Zhao, L. Zhang, Y. Liu, J. Luo, and J. Sun, "Understanding How the Requirements Are Implemented in Source Code," Proc. Asia-Pacific Software Engineering Conference, pp. 68-77, December 2003.