SNIAFL: Towards a Static Noninteractive Approach to Feature Location

WEI ZHAO and LU ZHANG
Peking University
YIN LIU
Rensselaer Polytechnic Institute
and
JIASU SUN and FUQING YANG
Peking University

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 article, we present a static and noninteractive method for achieving this objective. The main idea of our approach is to use information retrieval (IR) technology to reveal the basic connections between features and computational units in the source code. Due to the imprecision of retrieved connections, we use a static representation of the source code named BRCG (branch-reserving call graph) to further recover both relevant and specific computational units for each feature. A premise of our approach is that programmers should use meaningful names as identifiers. We also performed an experimental study based on two real-world software systems to evaluate our approach. According to experimental results, our approach is quite effective in acquiring the relevant and specific computational units for most features.

Categories and Subject Descriptors: K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, syntax*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Experimentation

This research was partially supported by the National 973 Key Basic Research and Development Program No. 2002CB312003, the State 863 High-Tech Program No. 2004AA112070, and the National Science Foundation of China No. 60125206, 60233010, 60373003, and 60403015.
This article is a revised and expanded version of a work presented at the 26th International Conference on Software Engineering in May, 2004.
Authors' addresses: W. Zhao, L. Zhang, J. Sun, and F. Yang, Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P. R. China; email: {zhaow,zhanglu,sjs,yang}@sei.pku.edu.cn; Y. Liu, Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY 12180; email: liuy@cs.rpi.edu. This research was performed while the third author was at Peking University.

Additional Key Words and Phrases: Program comprehension, feature location, traceability, information retrieval, static analysis, BRCG

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 Turver and Malcolm [1994], about 40 percent of the total cost of a software project is spent on software maintenance.

Usually, a maintenance task is to modify or add some functionalities or features [Yau et al. 1988; Bohner and Arnold 1996], or refactor the program without changing its behavior [Fowler et al. 1999]. Although some refactoring tasks (such as refactoring for generalization [Tip et al. 2003]) can be fulfilled automatically, most maintenance tasks require maintainers to spend more than half of their working time analyzing documents and the source code to understand the features of the system being maintained [Corbi 1989]. A basic but very helpful step for this kind of maintenance is to locate interesting features in the source code [Wong et al. 1999].

More theoretically, the feature location problem can be formulated as identifying the relationships between the user's view and the programmer's view [Wilde et al. 1992]. The user's view is made up of a collection of features denoted as $FEATURES = \{f_1, f_2, \ldots, f_n\}$, while the programmer's view consists of a collection of computational units denoted as $UNITS = \{u_1, u_2, \ldots, u_m\}$. Thus, the feature location problem is to recover the implementation relationships over $FEATURES \times UNITS$. In particular, two kinds of implementation relationships are usually of special interest [Wong et al. 1999; Eisenbarth et al. 2003]. 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 and 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 [Biggerstaff et al. 1993; Chen and Rajlich 2000; Griswold et al. 2001; Robillard and Murphy 2002]) can be used to assist maintainers to locate features. In the literature, this kind of approach is also referred to as the static approach. Secondly, automatic approaches based on dynamic execution of the system (e.g., [Wilde and Scully 1995; Wong et al. 1999]) are also reported in the literature. These approaches are usually referred to as dynamic approaches. The latest work on feature location by Eisenbarth et al. [2003] is a basically a dynamic approach, but also uses static analysis for refinement.

In this article, we propose a static noninteractive approach to feature location (SNIAFL). Like dynamic approaches, our approach works in a batch-like manner without much human involvement. However, unlike dynamic approaches which use test cases to exhibit the basic relationships between features and units, we use information retrieval (IR) to achieve this goal. In fact,

our approach is inspired by recent advances in applying IR for recovering traceability between code and documentation (e.g., [Antoniol et al. 2002; Marcus and Maletic 2003]). According to the characteristic of the retrieved results, we use the branch-reserving call graph [Qin et al. 2003] (an expansion of the call graph with branch information) to further recover the relevant and specific units, and acquire relationships among the relevant units for each feature.

The remainder of this article is organized as follows. In Section 2, we analyze related approaches to the feature location problem and discuss other related research. Section 3 overviews the SNIAFL approach to feature location and Section 4 presents this approach in detail. In Section 5, we report an experimental study applying our approach to two real-world software systems. Section 6 further discusses the advantages and disadvantages of our approach. Section 7 concludes this article.

2. RELATED WORK

2.1 Feature Location

As mentioned above, the central task of feature location is to match knowledge about features to that about computational units. In previous research, two mainstreams of ideas for this task can be identified. The first assumes that maintainers with 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. The second 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 test cases. Therefore, the feature location problem is turned into analyzing execution traces with feature tags. This leads to the various dynamic approaches.

The forerunner of interactive feature location is Biggerstaff et al. [1993], in which this problem is referred to as a concept assignment problem. It should be noted that a *concept* usually has a wider meaning than a *feature*. Thus, a feature can be viewed as a concept that is related to the executions using some specific input data. In Biggerstaff et al.'s approach, several graphical representations of the source code (such as the call graph and the program clustering graph) as well as some regular-expression-based matching tools are exploited to facilitate the process of assigning human-oriented concepts to program-oriented concepts. Similarly, another interactive approach to feature location is based on browsing the abstract system dependency graph (ASDG) [Chen and Rajlich 2000]. An ASDG can represent the dependencies among routines, types, and variables at an abstract level. It therefore can guide a user to search for the implementation of a particular feature. Griswold et al. [2001] report the aspect browser, which can help maintainers to find feature implementations using lexical searches. This tool is based on Seesoft [Eick et al. 1992] and uses the map metaphor to graphically represent the location of possible pieces of code for feature implementations. Concern graphs is proposed as a facility of

feature location in Robillard and Murphy's [2002] work, where features are referred to as concerns. Compared to previous interactive approaches, the main difference is that the building of the concern 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 latest publication of Marcus et al. [2004] presents an approach to locate the domain concepts of interest for a given system by applying one of the IR models (i.e., latent semantic indexing (LSI) [Deerwester et al. 1990; Baeza-Yates and Ribeiro-Neto 1999, 44–46]. As the concepts in it are very similar to the features discussed here, this work can be viewed as closely related to our work. Marcus et al. treat the concepts expressed in natural languages as queries and retrieve the corresponding source code artifacts through LSI for them. The retrieved results still need further processing manually. The query corresponding to the investigated concept is either manually generated by engineers, or automatically constructed by LSI based on one or more simple initial terms about it. Besides the human involvement, another difference between this approach and our work is that no syntactic or semantic information extracted from the source code is employed.

The main advantage of 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 locating it. However, the interactive nature of these approaches make them very difficult to be highly automatic, and intensive human involvement is unavoidable.

The pioneer work of dynamic feature location is software reconnaissance [Wilde et al. 1992; Wilde and Scully 1995]. In this approach, carefully designed test cases (amongst 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 identified: *commonly involved* units, *potentially involved* units, *indispensably involved* units, and *uniquely involved* units. A similar approach is reported in Wong et al. [1999], where the main difference is that it can help to identify 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. [2001, 2003] have published several articles using concept analysis for feature location. This approach uses a concept lattice to represent the execution traces recorded in dynamic execution. Based on the lattice, several different relationships between features and computational units can be easily recovered. It also uses static analysis based on the dependency graph to further refine the results achieved by concept analysis. Licata et al. [2003] present a notion of *feature signature* to investigate the properties of changes in evolving programs based on dynamic feature location. They indicate that most changes tend to pertain to either a very small number of features, or to almost all of them.

The main advantage of dynamic approaches is that they can automatically deal with many features in a batch-like manner after the test cases are obtained.

However, the design of test cases may be a difficult task and a large number of test cases are required for the scrutiny of investigated features. It should be noted that automatic test case generation techniques (such as [Gupta et al. 1998; Shan et al. 2001]) can hardly help here, as the test cases designed for feature location are aiming at acquiring test cases for each concerned feature and not just those for revealing faults.

Rajlich and Wilde [2002] summarize some static and dynamic approaches to feature location and discuss the role of concepts for program comprehension. A simple empirical comparison of a dynamic approach (i.e., software reconnaissance) and a static approach (i.e., the approach in Chen and Rajlich [2000]) is presented in Wilde et al. [2003]. 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 Chen and Rajlich [2000] 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. [2001, 2002] have published a series of articles on recovering code to documentation traceability. 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 [Baeza-Yates and Ribeiro-Neto 1999, 30–34] and the vector space model [Baeza-Yates and Ribeiro-Neto 1999, 27–30] are used in this approach without much difference in terms of performance. Marcus and Maletic [2003] use the latent semantic indexing (LSI, which is based on the vector space model) method for recovering 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, this approach can to some extent outperform Antoniol et al.'s approach. Marcus and Maletic [2001] have also used the LSI method to define similarity measures between source code elements.

In the context of document-to-code traceability recovery, if the document of interest is requirement documentation where each one or more paragraphs describe a feature, the constructed links can be viewed as the relationships between features and source code artifacts and therefore serve as a solution of feature location. Locating features only based on IR is usually very coarse and cannot cater to the request on precision to address this problem in practice. However, the use of IR does provide an automatic means for connecting human-oriented knowledge and program-oriented knowledge. This is the starting point of our approach.

Cubranic and Murphy [2003] apply information retrieval as well as other matching techniques to recover the implicit traceability among different kinds of artifacts of open source projects (i.e., source file revisions, change or bug tracks, communication messages, and documents). Hipikat, a corresponding tool, can recommend some software development artifacts to newcomers under

current tasks based on these recovered links. Similarly, CVSSearch, which is also for open source projects, constructs the connections between previous CVS comments and the changed code to facilitate current searching [Chen et al. 2001].

2.3 Graph-Based Pruning

In previous research on decision trees, several graph-based pruning methods have been investigated to acquire the right-sized decision trees. In our approach, as we are aiming at static noninteractive feature location, it is a natural idea to use graph-based pruning to discard irrelevant nodes and keep the relevant nodes in a static representation of source code. Therefore, the research on graph-based pruning is also related to our work, in terms of dealing with static representation of source code.

Typically, a pruning method for decision trees discards unnecessary branches of a given decision tree in a repeated manner until the required minimum size for the pruned tree is reached. The most popular and widely used pruning method for decision trees is the cost-complexity pruning (CCP) method introduced by Breiman et al. [1984]. During the course of each iteration of CCP, one candidate node is picked out among all subnodes of the root node of the previously-pruned decision tree and the candidate node is pruned including all its subnodes. The determination of the candidate node is based on a criterion which indicates the necessities of the nodes in a decision tree to make precise classifications. The dynamic-programming-based pruning (DPP) algorithm retrospects all previous pruning results and chooses a combination of candidate nodes to be pruned off the original decision tree each time [Li et al. 2001]. The aim of DPP is to construct a sequence of optimally pruned trees whose sizes are reduced one leaf at a time. The measurement of a single node of DPP is the same as that of CCP, but the criterion for determining the candidate nodes to be pruned off is different. Multiple nodes can be picked out and the pruning is always based on the original decision tree.

3. AN OVERVIEW OF THE APPROACH

Since our static noninteractive approach to feature location (SNIAFL) performs as a batch-like method, each step of the approach refers to a particular model or algorithm. Before describing details, we overview the objectives and main ideas of our approach in this section first. A preliminary application of a similar idea can be also found in our previous work [Zhao et al. 2003].

As our approach is currently evaluated on software systems written in the C programming language, we hereafter use the term *function* instead of *computational unit* to present our approach. As mentioned above, the feature location problem recovers the implementation relationships between features and functions. In this article, we concentrate on locating *relevant* functions and *specific* functions of a feature, although other relationships between features and functions can also be acquired via a slight extension of our approach (which is discussed at the beginning of Section 4.4). In detail, *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; and 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 each feature and can be computed through comparing all features' relevant functions.

The goal of our approach is to solve the feature location problem statically while avoiding extensive human interaction. To achieve this goal, the basic idea is to use IR as a means to reveal the basic connections between features and functions. As indicated by recent studies on applying IR to recover traceability links, IR may be quite effective in this kind of task. Obviously, the potential advantage of this idea is that it can save the costs for creating and executing test cases in dynamic approaches, and those for human involvement in interactive approaches. Like approaches to traceability recovery [Antoniol et al. 2000, 2002; Marcus and Maletic 2003], our idea also 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 IR technology, we cannot always acquire an accurate set of relevant functions for a feature directly from IR. This means that some irrelevant functions may be included due to the use of some nonspecific words in feature descriptions, while some relevant functions may be excluded due to the lack of corresponding descriptions in features. As a result, we have to aim at correctly filtering some specific functions for each feature when using IR since the specific descriptions of each feature, as the essential part, should not be lost and can be strengthened through some IR models according to the characteristics of their distributions. To achieve this, we use an IR model to decrease the importance of nonspecific words related to nonspecific functions, and for each feature we acquire a function list ranked by the extent to which the function is specific to the feature. Then, an algorithm is used to set a division point in the list according to the distances among ranked functions. All functions before this division point will be used as the initial specific functions to the 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 the initial specific functions revealed by IR 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. As no information about call-relationships between the retrieved functions can be acquired from IR, it is also helpful to recover this information for maintainers to understand how the relevant functions are connected with one another. 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 was originally proposed for discovering use cases from source code in Qin et al. [2003]. Compared with the traditional call graph, the branching information in this representation can be used for eliminating irrelevant functions and refining call-relationships. With branching information as well as sequential information in the call graph, the pseudoexecution traces for each feature can

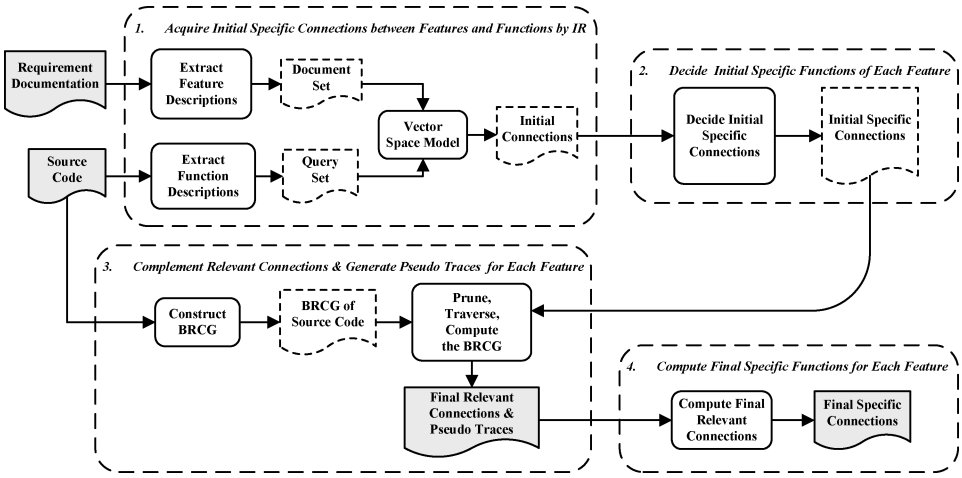


Fig. 1. Process of the SNIAFL approach.

be generated. 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 STATIC NONINTERACTIVE APPROACH TO FEATURE LOCATION

The process of SNIAFL approach is depicted in Figure 1. Corresponding to basic ideas behind the approach presented above, SNIAFL consists of four main steps:

(1) Acquisition of initial specific connections between features and functions. In this step, we use an IR model to filter specific information of the features and recover the initial connections between features and functions with this information.

(2) Determination of initial specific functions related to each feature. The second step is to rank functions for each feature according to the initial retrieved results, decide a separation point, and choose the preliminary specific functions for each feature for further refinement.

(3) Acquisition of relevant functions and possible pseudoexecution traces for each feature. Based on each feature's initial specific functions, we prune those branches which are mutually exclusive from those where there is at least one initial specific function based on the BRCG extracted from source code. The functions in the pruned BRCG are regarded as relevant functions for each feature, finally. As the BRCG maintains the branching and sequential information of source code, the possible pseudoexecution traces of these relevant functions can also be generated.

(4) Computation of final specific functions for each feature. In the last step, we analyze relevant functions to determine the final specific functions according to the definition of specific functions.

In the next four subsections, we describe the above four major steps in detail. During the course of the presentation, we use examples to illustrate each step's

Feature 1: Pops two values off the stack, adds them, and pushes the result. The precision of the result is determined only by the values of the arguments, and is enough to be exact.

Feature 2: Pops two values, subtracts the first one popped from the second one popped, and pushes the result.

Functions:

main
dc_string_init
getopt_long
dc_evalstr
dc_func
dc_pop
dc_push
dc_binop
dc_add
dc_sub
dc_top_of_stack
dc_register_pop
dc_register_get

Fig. 2. Example features and functions.

processing. Actually, the examples are derived from one of our studied systems (the DC system, an implementation of a reverse-polish desk calculator). All the data is from real experimental results, although for simplification in some cases we only present the results partially.

The example features' descriptions and a subset of all functions of the DC system are shown in Figure 2. Two features of DC are used to illustrate our approach in the following subsections. A subset of all functions of DC is chosen for ease of illustration.

4.1 Acquiring Initial Specific Connections Between Features and Functions

In the first step, as shown in Figure 1, the requirement specification and source code of a certain system under consideration are the input for preparing queries and documents processed by IR. The output is the initial specific connections between the features and functions of the system constructed by IR, as well as the quantified degree of similarity for every connection.

4.1.1 *The Vector Space Model of IR.* In our approach, we use the vector space model of IR for indexing documents and queries and ranking the results. Here, we briefly introduce the vector space model. Please refer to Baeza-Yates and Ribeiro-Neto [1999, 27–30] for details.

The vector space model [Salton and Lesk 1968; Salton 1971] proposes a framework in which partial matching is possible. It treats queries and documents as vectors constructed by 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 query. Vector $\vec{q}\ (q_1, q_2, \ldots, q_t)$ represents the query q, in which q_i is the weight of the ith index term in the query q, and t is the number of index terms. Vector $\vec{d_j}\ (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$ 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 index terms. The vector space model evaluates the degree of similarity of the document d_j with regard to the query q as a correlation. This correlation can be quantified

by the cosine of the angle between two corresponding vectors (i.e., $\vec{q}$ and $\vec{d_j}$, which is shown in equation (1).

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

In order to compute the degree of similarity using equation (1), it is necessary 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).

$$f_{i,j} = \frac{freq_{i,j}}{\max_l freq_{l,j}} \tag{2}$$

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 document d_j); the maximum is computed over all index terms that are mentioned in the text of the document d_j; and $f_{i,j}$ is the normalized frequency of the ith index term in the document d_j.

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

$$idf_i = \log \frac{N}{n_i} \tag{3}$$

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 using the *idf* factor is that terms appearing in more documents are less useful for distinguishing a relevant document from a nonrelevant one.

Then, as suggested in Baeza-Yates and Ribeiro-Neto [1999, 27–30], the two following equations (i.e., (4) and (5)) can be used to compute the weights of index terms in documents and queries with *tf* and *idf* factors.

$$w_{i,j} = f_{i,j} \times idf_i \tag{4}$$

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

4.1.2 *Preparing Queries and Documents.* The decision to treat feature descriptions as documents and function descriptions as queries for IR is an essential part of our approach. In this section, we first describe how to prepare the feature description set and function description set, since no matter which one is treated as query, the preparation remains the same. After that, we present the rationale of our decision.

The set of feature descriptions (the document set) can be acquired from requirements documentation, domain experts, or even users familiar with the

target system in the case where there is no available requirement documentation. 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). The examples are shown in Figure 2, which are derived from the user manual in the DC package. 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 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 infinitive form of verbs) to be the final index terms.

The function description set (the 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, and 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 into the body of the function (such as local variables, which were never considered in our previous work [Zhao et al. 2003] and show less contributions in the retrieval step). 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*. It should be indicated that this relatively simple preprocessing is not enough for further use of IR. For instance, identifiers like *featurelocation* and *floc* need some more sophisticated word recognizers. In our experiment, we preprocess such cases manually although some slight extension using existing word-extraction techniques (e.g., [Anquetil and Lethbridge 1998]), and prefix matching may help. After preprocessing, the words obtained from the identifiers will be transformed into a set of index terms using the same rules as above used for transforming requirement descriptions.

In the following, we present the rationale for the treatment of feature descriptions as documents in the IR step. From the 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 degree of similarity between them. The *tf* and *idf* factors are used to measure the weights of index terms in the query and in each document. As described in Section 4.1.1, computations of the *tf* and *idf* factors are based on statistical data of the query and documents. For the *tf* factor, the more appearances of one index term in a certain document or query, the higher the weight of this index term in this document or query. For the *idf* factor, the more appearances of one index term in all the documents, the less contribution the index term makes to judging the similarity between the query and each document.

As we are aiming at retrieving some specific functions for each feature, we can apply this *idf* factor to filter specific information of the feature descriptions if we treat the features as documents in the IR step. Thus, common descriptions in different features will contribute less to computing the similarity between features and functions, and the ranking will be mainly based on the specific

Table I. The Retrieved Initial Connections between Example Features and Functions

Feature 1	dc_add	0.4153
	dc_pop	0.2159
	dc_top_of_stack	0.2054
	dc_register_pop	0.1869
	dc_register_get	0.1554
	dc_push	0.0289
Feature 2	dc_sub	0.7726
	dc_pop	0.2909
	dc_top_of_stack	0.2515
	dc_register_pop	0.2431
	dc_register_get	0.1976
	dc_push	0.0525

descriptions in the features. That is why we decide to treat the feature description set as documents, and the function description set as queries in our approach.

The two example features in Figure 2 have index terms *pop* and *push*; and there are corresponding functions with these index terms (such as *dc_pop*, *dc_register_pop*, *dc_push*, and *dc_register_push*). These common index terms should contribute less during the construction of connections via the effect of the *idf* factor. On the other hand, those specific index terms (such as *add* and *subtract*) should be filtered with higher importance for recovering the relations between features and functions. That is to say, functions *dc_add* and *dc_sub* should relate to feature (1) and feature (2), respectively, with higher rank values than functions *dc_pop*, *dc_register_pop*, *dc_push*, and *dc_register_push*. Thus, by treating feature descriptions as documents, we are more likely to achieve our objective of ranking those specific connections with higher values.

4.1.3 *Retrieving Initial Connections.* After both the query set and the document set are prepared, we use the vector space model for 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. These retrieved documents can reveal all the connections between the function and all of 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 by transposing the retrieval results. For example, there are n features in the feature set $F = \{f_1, f_2, \ldots, f_n\}$ and m functions in the function set $U = \{u_1, u_2, \ldots, u_m\}$. The similarity value between f_i and u_j is S_{ij} ($1 \leq i \leq n$, $1 \leq j \leq m$). The original retrieval result for function u_j is $\{f_1, f_2, \ldots, f_n\}$ with rank values $S_{1j}, S_{2j}, \ldots, S_{nj}$. The acquired result for feature f_i is then $\{u_1, u_2, \ldots, u_m\}$ with rank values $S_{i1}, S_{i2}, \ldots, S_{im}$.

After the first step, for the example features and functions in Figure 2, the recovered connections with rank values are shown in Table I. The rank values

Input: *u, m* (where *u* is the function array in
descending order, *m* is the number of this array)
Output: *S* (the initial specific function set)
Step 1: $S \leftarrow \varnothing$
Step 2: **for** $i = 2$ **to** m
$d[i\text{-}1] \leftarrow u[i\text{-}1].rankvalue\text{-}u[i].rankvalue$
Step 3: $dmax \leftarrow d[1]$
$divisionpoint \leftarrow 1$
for $i = 2$ **to** $m\text{-}1$
if$(d[i] > dmax)$
$divisionpoint \leftarrow i$
Step 4: **for** $i = 1$ **to** *divisionpoint*
$S \leftarrow S \cup \{u[i]\}$

Fig. 3. Algorithm to determine the division point and choose the initial specific functions.

are computed based on all features in our experiment. That is to say, the *idf* factor is based on all of DC's features, rather than on only these two example features. From Table I, we can see that functions *dc_add* and *dc_sub* have much higher rank values than function *dc_pop*, since they correspond to the filtered specific descriptions rather than to the common descriptions due to the *idf* factor.

4.2 Identifying Initial Specific Functions

After acquiring the initial connections between features and functions, 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 all functions in the list. We simply use the arithmetic differences of the rank values of the functions as the distances between them. We regard 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 the others are not. It is obvious that the functions before this point have much closer distances and, therefore, are more likely to have the same nature (i.e., the specific nature) as the feature. The algorithm to determine the division point and to choose the initial specific functions is depicted in Figure 3.

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

The first step is to initialize the output initial specific function set S. Step (2) calculates all distances $d[i]$ $(i = 1, 2, \ldots, m - 1)$ between two consecutive functions sorted in descending order. In the third step, for all acquired distances the biggest one is chosen and thus, 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.

According to the rank values of the two example features in Table I, the calculated division points of Feature (1) and Feature (2) are both after the first function, since the distance between the first function and the second is biggest among all distances between two consecutive functions. Therefore, functions *dc_add* and *dc_sub* are the initial specific functions for the two features, respectively.

4.3 Determining Relevant Functions and Generating Pseudoexecution Traces

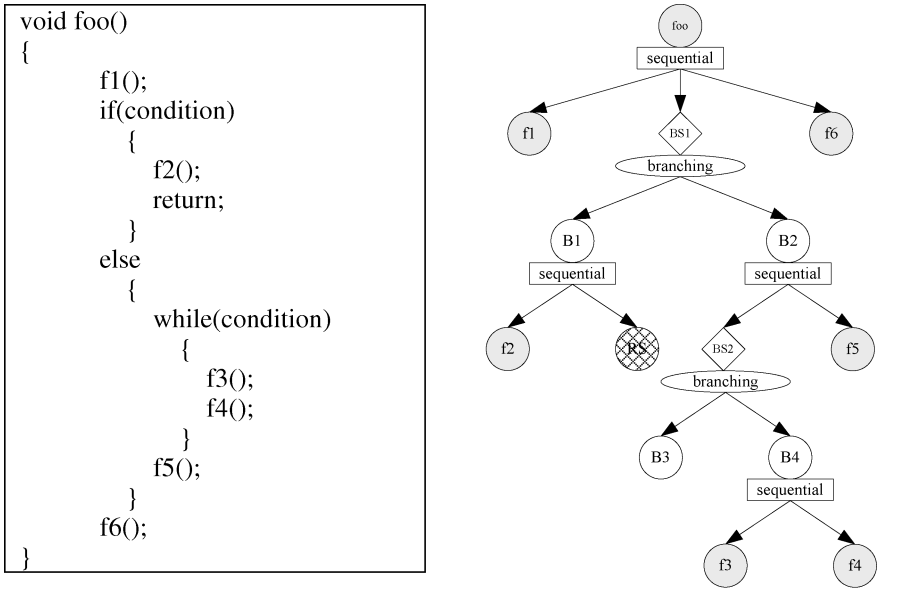
After the initial specific functions are identified, we begin to determine the relevant functions and to generate the pseudoexecution traces of these functions. The basis of this step is obtaining the BRCG from the source code. We traverse the BRCG and determine the relevant functions according to the initial specific functions, and finally, construct the possible execution traces via the structural information among these functions in the BRCG.

4.3.1 *The Branch-Reserving Call Graph.* The branch-reserving call graph (BRCG) was first introduced in Qin et al. [2003] for discovering use cases. This structure is a representation of the source code by adding branch information to the traditional call graph. In this structure, both branch statements and function-call statements are considered. For simplicity, *if*-statements, *case*-statements, and *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 connections between the nodes are the sequential control flow and the branching control flow. Therefore, the BRCG can be formally defined as 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 control flows, 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 control flows, 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 BRCG is extended from the traditional call graph. The first three conditions further define that the BRCG is a call graph with two kinds of control flows (i.e., sequential control flow and branching control flow). The fourth condition states that a node connects to (i.e., invokes) its subnodes only through one kind of control flow. An example BRCG is depicted in Figure 4. Figure 4(a) depicts the source code of a function *foo*, and Figure 4(b) depicts the corresponding BRCG.



(a) Sample code for BRCG (b) A sample BRCG

Fig. 4. A branch-reserving call graph.

The construction of the entire BRCG for a system is based on the subBRCGs of all the functions. The entire BRCG of a system can be acquired by connecting all the subBRCGs for the functions into complete graph by linking the root node of each subBRCG to the nodes of the same signature in other subBRCGs. Please refer to Qin et al. [2003] for the algorithm to build the BRCG for each function from source code. Identical to Qin et al.'s strategy, we do not include the edges of recursive function calls in the BRCG for simplicity, since it will not affect the approach much.

4.3.2 *Acquiring Relevant Functions Using the BRCG.* Since the specific functions of a feature mean that the implementation of this feature will definitely invoke these functions and implementation of other features will not invoke them, it is very likely that branches that will not invoke any such functions are irrelevant to the feature. Therefore, we can prune some branches from the BRCG according to the nonexistence of the initial 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 $BS_i\_b_2$ does 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 propagated to $BS_j\_b_1$. This propagation will end at the root node of the BRCG. This is the main idea we use to prune the BRCG. Obviously, all the functions left in the pruned BRCG should be candidates for the relevant functions to the feature.

Since there may be more than one initial specific function for a feature, pruning is a repeated process based on each initial specific function. We process the initial specific functions in descending order based on their rank values against the feature. The pruning of each initial specific function is based on the previous pruning result. If the initial specific function under consideration currently does not exist in the BRCG acquired in the previous pruning step, it probably means it is a wrongly-retrieved result. In such a case, this function will not be considered any further.

The algorithm for pruning the BRCG based on one particular initial specific function is depicted in Figure 5. The input to this algorithm is the BRCG with root node n, and the initial specific function *isf*. The output is the pruned BRCG.

This algorithm includes three main steps. The first step traverses the input BRCG and adds all the appearing positions (i.e., call sites) of the initial specific function to the set F. Since this step is a basic graph traversal problem, we do not discuss the details. The second step marks all branches where the initial specific function occurs. The process of propagating specific branches ends at the root node of the BRCG. Obviously, the first two steps can be accomplished at the same time. That is to say, once locating an appearance of the initial specific function, the specific link to its superbranch statements will be created and propagated up until the root node. We describe them as two separate steps, just for clarity. The third step prunes those unmarked branches that have the same branch statement as the marked branches. The definitely-used nodes and their subnodes will always be maintained in the pruned BRCG. This step is recursive, and it traverses the BRCG from the root node with a depth-first strategy. When the node is a branch statement and there is at least onespecific branch of it, all the nonspecific branches will be deleted. As the propagation of the specific relation in step (2) marks all specific branches, it is not needed to further traverse all the transitive descendents for each branch statement to determine the specific branches during the pruning processing. 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 considered as the relevant functions of the feature after the pruning according to all the initial specific functions. The worst case time complexity of this algorithm is linear in the number of nodes in the BRCG.

Considering the example BRCG in Figure 4(b), supposing function $f3$ is one of the initial specific functions for a certain feature, applying the pruning algorithm in Figure 5 on function $f3$ and this BRCG, then branches $B3$ and $B1$ are deleted. Since the branch $B4$ where function $f3$ occurs keeps a specific relationship to its branch statement *BS2*, the mutually exclusive branch of branch $B4$ (i.e., branch $B3$) is deleted. Furthermore, the specific relationship is propagated upward, so that branch $B2$ and its branch statement *BS1* hold it. Thus, branch $B1$ is pruned. As a result, function $f2$ is pruned and the relevant functions are: *foo*, $f1, f3, f4, f5, f6$.

4.3.3 *Generating Pseudoexecution Traces.* As the pruned BRCG includes both branching and sequential information, we can generate each possible execution trace by traversing the graph. For the relevant functions acquired in the

Input: a BRCG with the root node *n*, one initial specific function *isf*.

Output: the pruned BRCG.

Step 1: locate all call sites of the function *isf* in BRCG *n* to *F*

Step 2:

if $F = \varnothing$ **return** *n*

for each f_i in set *F* (i = 1, 2, ..., *size*(*F*))

  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 3: traverse the BRCG marked by specific relation from *n*

currentnode := *n*

if *currentnode* is a leaf node

  return *n*

if *currentnode* is a function

  traverse all its sub nodes as step 3

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 3

  end

Step 4:return *n*

Fig. 5. Algorithm to prune the BRCG according to the initial specific function.

pruned BRCG, we use the following algorithm to generate the possible execution traces. We refer to them as pseudoexecution 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). Figure 6 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: $S := \varnothing$

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

$T := \varnothing$

for each trace t in S_i

for each s in U and s does not end with "RETURN"

$T := T \cup \{s+t\}$

$U := T$

end

if n represents a function

for each s in U

$S := S \cup \{s+$"-EXIT_"+n.label$\}$

else

$S := U$

end

Step 5: **if** the relationships between n and its sub-nodes are branching

for each S_i

$S := S \cup S_i$

Step 6: **return** S

Fig. 6. Algorithm to generate pseudoexecution traces.

```
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. 7. Algorithm to determine the specific functions.

This algorithm is a recursive one, during which we only generate trace information for nonleaf nodes that represent functions and all the leaf nodes. The generated pseudoexecution traces can somehow reveal the relationships between the relevant functions.

4.4 Determining Specific Functions

After we acquire all the relevant functions of each feature from the pruned BRCG, we can determine all other relationships between features and functions in the same way that a dynamic approach deals with the recorded functions invoked for each feature. Due to space limitation, we present only the algorithm for calculating the final specific functions for each feature, which is depicted in Figure 7. This algorithm is based on the relevant relationships between the features and functions. We construct a two-dimensional 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-dimensional array. $V[i,j]$ equals 1 if the jth function is the relevant function to the ith feature, otherwise, it is 0.

The idea of this algorithm is to check each feature like so:

(i) It finds the first function that is relevant to this feature;
(ii) It checks whether this function is also relevant to another feature;
(iii) If not, it marks the function to the feature with 1 in the output array, and otherwise 0;
(iv) It finds another relevant function, and does the same;

(v) After all the functions are processed, the specific functions for one feature can be acquired.

Obviously, the worst case time complexity is $O(n^2m)$, where n is the number of features and m is the number of functions.

5. AN EXPERIMENTAL STUDY

In our experiments we used two software systems of GNU, named DC [GNU DC] (which is distributed with the BC package) and UnRTF [GNU UnRTF]. We acquired the complete source code and the requirements specification documentation of these two systems. For DC, there were 21 primitive features, each of which implemented a single functionality of the system. The others were composite features. Their location could be acquired through the analysis of all the locations of their constituent features. For the UnRTF system, there were eight primitive features, among which six features were implemented at level of functions. In the following subsections, firstly we present our experimental method and then report the results of the applications of our approach on these two systems. In our experiment, we only studied the location of these 27 features.

5.1 Experimental Method

To apply our method, we used SMART [Salton 1971] as the tool in the IR-step. SMART is an implementation of the vector space model of IR proposed by Salton back in the 1960'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 27 features of these two systems, we applied SNIAFL to get the relevant functions, the possible pseudoexecution traces, and the specific functions. For each feature, we got three groups of data for 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 the third is the pseudoexecution traces of these relevant functions constructed by the BRCG.

To evaluate SNIAFL, we manually analyzed the two systems, and for each of the 27 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 the instrumented versions of these two systems, in order to get the dynamic results. As dynamic approaches are heavily dependent on the quality of the test cases, they will not be complete and/or precise if the test cases are not sufficient or well designed. Therefore, for each feature we designed two groups of test cases to imitate both the test cases made by an experienced maintainer and those made by a typical maintaining engineer in the real world. Each test case invoked exactly 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 Figure 7. To confirm the necessity of using the function-to-feature

retrieving strategy and the BRCG in our approach, we also recorded the retrieval results using features as queries and functions as documents as the relevant functions. This is referred to as the IR-only approach in this article, in which 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 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 Experiments on DC

The DC system is a reverse-polish desk calculator which supports unlimited precision arithmetic calculation. It consists of about 2.7KLOC in ANSI C programming language, and 74 functions. Among the 49 features of the DC system, we experimented on the 21 primitive features according to the experimental method presented above. The results on this system are reported in the following.

5.2.1 *Results on Acquiring Relevant Functions. Quantitative Results*—We calculated the *precision* and *recall* of the relevant functions acquired by the three approaches on the DC system. Table II shows the results of the three approaches for the 21 features. Since the functions invoked by the test cases for any feature should be relevant to the feature (supposing there are no wrongly-involved test cases for feature location), the *precision* of the dynamic approach is always 100 percent. Therefore, we do not list it in the table.

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 precondition is that the test cases are well designed. The recall of the dynamic approach with insufficient test cases is much lower than that of 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, the recall is 99.57% and the 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 in acquiring the relevant

Table II. The Recall and Precision of Relevant Functions of Three Approaches on DC

	IR Only		Recall of Dynamic Approach		SNIAFL	
Feature No.	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%

functions to some extent. However, we still have an exceptionally bad case in which the precision is only 37.5%.

Qualitative Analysis—The bad performance of the IR-only approach on the DC system 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's description. Therefore, they cannot be acquired by the IR-only method. For the insufficient precision, the explanation lies in the feature-to-function retrieval strategy. Some commonly used words in the description of a feature will lead to retrieving all the functions having some of those words as identifiers. The dynamic approach is much more precise. Its only disadvantage is its 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 treat more than the 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 our inability to reach total correctness. The precision of our approach is not as good as the recall. This is due to the conservative nature of the static approach that takes all possibilities into consideration. Especially in the worst case (feature 21), the precision is down (to 37.5%). In this case, the semantic information of the program does significantly affect the implementation of this feature. Our approach cannot get such information and therefore collects some functions in branches that are

Table III. Execution Traces on DC

Feature No.	Dynamic Approach		SNIAFL		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

syntactically relevant but semantically irrelevant. This situation is discussed in more detail in Section 6.5.

5.2.2 *Results on Acquiring Function Relationships. Quantitative Results*—Due to the simplification of the loop statement, the pseudoexecution traces acquired from the BRCG are not exactly the same as the traces from dynamic execution in some cases. For these cases, in our experiment we still treat the acquired traces as the correct ones as long as they can reveal the actual execution relationships.

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

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 well-designed test cases, there are still some traces lost, while for the insufficient test cases, more traces are lost. Except for 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.

Qualitative Analysis—Our approach is not quite effective in acquiring execution traces for each feature. However, it 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.

5.2.3 *Results on Acquiring Specific Functions. Quantitative Results*—Table IV shows the results of initial and final specific functions of our approach,

Table IV. Specific Functions on DC

Approaches		Totally Correct	Totally Wrong	Partially Wrong	Correct Ratio
SNIAFL	Initial	14	3	4	66.67%
	Final	18	3	0	85.71%
Dynamic	Insufficient	20	0	1	95.24%
	Well-designed	21	0	0	100%

and the results of the dynamic approach with insufficient and well-designed test cases in the experimentation. Due to space limitations, we concisely present the results in Table IV. The column captioned by *totally correct* indicates the number of features which acquire all specific functions without additional false results. Features with no correct specific function are enumerated in the column marked *totally wrong*. Features with correct specific functions as well as wrong ones are considered as those acquiring the *partially wrong* results. The correct ratio is computed based on the number of totally correct features over all features.

Of all 21 features in our experiment, 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 an 85.71% correct ratio of the specific functions after analyzing the relevant functions. This ratio is quite acceptable in practice.

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

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 for those partially-wrong initial specific functions. The explanation is that we can eliminate the nonspecific functions from the partially-wrong initial specific functions and complement those supporting ones. However, for those that are totally wrong, we can do nothing at this stage.

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

5.3 Experiments on UnRTF

The UnRTF system is a batch processing program to convert RTF (rich text) documents to other formats. UnRTF consists of about 8.6KLOC in ANSI C programming language, including comments, and 154 functions. There are eight primitive features of UnRTF, and two of them are implemented at the statement level rather than the function level. Consequently, we experimented only on the six primitive features. An example feature of the UnRTF system is "*Converting*

Table V. The Recall and Precision of Relevant Functions of Three Approaches on UnRTF

	IR Only		Recall of Dynamic Approach		SNIAFL	
Feature No.	Recall	Precision	Insufficient	Well-designed	Recall	Precision
1	9.66%	93.33%	57.93%	99.31%	100%	98.64%
2	11.03%	94.12%	57.93%	99.31%	21.38%	100%
3	11.03%	88.89%	57.93%	99.31%	100%	97.97%
4	9.66%	93.33%	57.93%	99.31%	100%	98.64%
5	9.66%	93.33%	57.24%	98.62%	100%	98.64%
6	9.66%	93.33%	57.24%	98.62%	100%	98.64%
Avg.	10.12%	92.72%	57.7%	99.08%	86.90%	98.76%

to Text with VT100 control codes." It is obvious that the features' descriptions of UnRTF are relatively concise compared with those of the DC system. The experimental results on the UnRTF system are as follows.

5.3.1 *Results on Acquiring Relevant Functions. Quantitative Results*—Identical to experiments on the DC system, we calculated the *recall* and *precision* of the relevant functions of the six features of UnRTF acquired by IR-only, both for the dynamic approach and our approach. The results are shown in Table V.

From the results in Table V, we can see that, unlike the results of DC acquired from the IR-only approach, the precision for the UnRTF system is much higher. Even the worst case is up to 88.89%, and the average is 92.72%. The recall of the IR approach is similar to the DC system.

For the dynamic approach, the recall keeps similar behavior to the DC system. The results of the dynamic approach on UnRTF are clearly affected by the quality of the test cases.

There is an abnormal case of our approach in acquiring the relevant functions for the UnRTF system. That is, for feature (2), the recall is only 21.38%. Except for this, the overall results show the effectiveness of our approach in acquiring the relevant functions for the features.

Qualitative Analysis—The better precision of the IR-only approach on the UnRTF system is due to the precise descriptions of the features. Unlike the DC system, there are no misleading and/or common words in the descriptions of features that may cause retrieval of irrelevant functions.

Unlike that abnormal case of the DC system, the unstable case of our approach on the UnRTF system (feature (2)) is not due to the lack of semantic information but to a false initial specific function. This function is indeed a relevant function for this feature, but not a specific function. At the same time, there are many relevant functions of this feature that appear in different branches of the same branch statement with that false specific function. The pruning process eliminates these functions such that the recall decreases seriously.

5.3.2 *Results on Acquiring Function Relationships.* The UnRTF application is a bit different from the DC system. For each feature that converts an RTF file to a certain file format, there are still many different branches to support the processing of different tags in RTF files. The possible combinations of different tags in the real input files cause the different execution traces. However,

Table VI. Specific Functions on UnRTF

Approaches		Totally Correct	Totally Wrong	Partially Wrong	Correct Ratio
SNIAFL	Initial	4	0	2	66.67%
	Final	6	0	0	100%
Dynamic	Insufficient	6	0	0	100%
	Sufficient	6	0	0	100%

such potential combinations make it difficult to compute the number of genuine execution traces. Consequently, we did not show the quantitative comparison for UnRTF in acquiring the execution traces. Nevertheless, our strategy that considers all the possible combinations of the different branches did acquire all possible pseudoexecution traces, based on the correct relevant functions. In these cases, we acquired far more than 10,000 execution traces. Such a large number weakens the effectiveness of our approach on generating the traces, although the recall is still high. Furthermore for feature (2), since not all the relevant functions were recovered by our approach, we still missed some genuine traces despite the large number of pseudoexecution traces.

5.3.3 *Results on Acquiring Specific Functions. Quantitative Results*—The results on acquiring the specific functions for each feature of the UnRTF system are shown in Table VI. We still compared the initial and final results of our approach with the dynamic approach (both the insufficient and the well-designed test cases). The meaning of the columns is the same as in Table IV.

Except the initial result of our approach with the 66.67% correct ratio, all achieve the 100% correct ratio on the UnRTF system. These results are encouraging for our SNIAFL approach, since, unlike the dynamic approach, it does not require test cases.

Qualitative Analysis—The good performance of our approach in acquiring the specific functions of the UnRTF system is attributable to two factors. Firstly, the precise descriptions of each feature did not introduce many irrelevant functions. Secondly, the effectively-recovered relevant functions finally exclude those false initial specific functions.

5.4 Summary

In the two experiments reported here, the experiment on DC has already been reported earlier [Zhao et al. 2004]. The experimental results on both target systems show clearly the overall effectiveness of SNIAFL in acquiring the specific functions and relevant functions for most features.

A major concern about the application of information retrieval is that it may cause the results to depend on the quality of documents (i.e., the descriptions of features). In our experiments, the features' descriptions of the UnRTF system more precisely express the features than do those of the DC system. From the experimental results, we can see that it does affect the results of the IR-only method (see the second columns in both Table II and Table IV) in the study. However, the impact on the overall performance of SNIAFL is not obvious. Actually, except for the abnormal cases of feature (2) in UnRTF and feature

(21) in DC in acquiring the relevant functions, SNIAFL has achieved acceptable performance for all features. The explanation lies in that suitable refinements on both IR- and static analysis steps can offset the impact of the coarse nature of information retrieval. However, if completely wrong initial connections are constructed by IR, the following steps in SNIAFL can hardly get the correct results. This is basically a main cause for the abnormal cases in the study.

The acquisition of pseudoexecution traces is not as effective as acquiring specific and relevant functions, since the large number of generated unnecessary combinations of relevant functions may offset the merit of the real ones. However, efforts to address this aspect may be of particular use in the following two circumstances. Firstly, some traces with abnormal processing branches of a certain feature can be easily acquired by SNIAFL, while dynamic approaches may have difficulties reaching them. Secondly, for those systems in which it is costly to carry out dynamic approaches, pseudoexecution traces of SNIAFL are an acceptable compromise for a simulation analysis.

5.5 Threats to Validity

The main threat to validity is to what extent the experimented systems are representative of all the possible target systems in practice. Although DC and UnRTF are real-world systems for various versions of UNIX and Linux, their sizes are still small compared to typical systems in practice. This threat can be reduced via experimenting on both 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 remain so in the eyes of professional testers. In our study we have tried our best to design these test cases, and the experimental results also confirm the effectiveness of the 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 can cause inconvenience in practice. Furthermore, the noninteractive way of our approach will prevent a maintainer from building up the knowledge about the feature in the locating process. However, we think the gains from the automatic nature can offset the price by saving much human involvement.

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

A main weakness of our approach is the lack of flexibility in choosing granularity. Using IR to set up connections between features and functions determines that the granularity has to be above the function level. There are circumstances in which 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 other dynamic approaches, there is much more flexibility for choosing functions, branches, or statements as the basic computational units.

6.4 Scalability

IR is a kind of technology that has been widely used in domains involving large numbers of data, such as web search. Compared with these domains, the descriptions of features and computational units should be viewed as only a small number of data, even if the target system is a very large industrial one. Therefore, the scalability of the IR step should not be a big concern. Furthermore, as scalability is always a heavily focused issue in IR, we can always apply cutting-edge techniques for improving the scalability of IR in our approach.

Another concern regarding the scalability of SNIAFL lies in the static analysis step. The construction of the BRCG is based on parsing the source code, in which only limited syntactic information is taken into consideration. Therefore, the complexity should not be worse than for a complete parsing for compilation. For pruning the BRCG, our approach will traverse the BRCG representation to prune the irrelevant nodes. As there is no iteration in the pruning process, its complexity should be proportional to the product of the number of initial specific functions and the number of nodes in the BRCG. For the calculation of final specific functions, the complexity should be proportional to both the square of the number of features and the number of functions.

The major concern about scalability of SNIAFL may be the preprocessing of identifiers using abbreviations and/or acronyms. Currently, there is some human involvement, and it may become a big burden when dealing with a very large system. This is also an issue we will pursue in the future.

6.5 Imprecision of BRCG

In order to make static representation suitable for computer processing, we try to avoid using the complex representation used in some other static approaches. However, there is some imprecision in our BRCG representation. A main shortcoming of BRCG is that it does not analyze the relationship between conditions in different branch statements. Considering the piece of code in Figure 8, there are two branch statements, each having two branches. In BRCG, these two branch statements are considered independent. However, this may not be the case in reality. For example, if branch one does not change the

```
if (x>1)
      {//Branch One
         ...
      }
else
      {//Branch Two
         ...
      }
if(x<-1)
      {//Branch Three
         ...
      }
else
      {//Branch Four
         ...
      }
```

Fig. 8. An example of the imprecision of BRCG.

value of x, executing branch one will exclude the execution of branch three. In our experience, this is another main reason for the imprecision in the SNIAFL approach (feature (21) of the DC system). We think that this situation may be improved if we can further apply some techniques of data flow and/or control flow analysis.

7. CONCLUSIONS AND FUTURE WORK

Feature location has long been identified as a necessary and helpful step in software maintenance. Although there have been many articles discussing this topic in the literature, most of the proposed approaches rely on manual test case generation and/or human analysis of source code for this purpose. Due to the recent advances in IR-based traceability recovery, we think that it may be advantageous to combine IR with static program analysis for feature location. This idea actually follows the direction pointed by some previous research (e.g., [Antoniol et al. 2000, 2002; Marcus and Maletic 2003; Marcus et al. 2004]).

Based on this idea, we have proposed a static and noninteractive approach to locating features, namely, SNIAFL, which combines the vector space information retrieval model and static program analysis based on an abstract representation of source code (i.e., BRCG). The starting point of our approach is to construct the initial connections between features and computational units in source code through matching the knowledge of features to the clues in source code using IR. This can be viewed as simulating first step in the process of manual feature location. Based on the initial connections, we recover all relevant computational units for each feature through navigating a static representation of the source code using some algorithms. This can be viewed as simulating human reasoning about the relationships between features and computational units in manual feature location. Thus, compared with manual feature location, SNIAFL can avoid much human involvement through combining IR and static analysis.

An experimental study on two GNU systems (DC and UnRTF) is also reported in this article, and our approach is evaluated against the IR-only method

and a dynamic approach from three aspects, respectively—relevant functions, pseudoexecution traces, and to specific functions. The experimental results are quite promising, as SNIAFL performs much better than the IR-only method on average, and it even achieves comparable results the dynamic approach in most cases. Therefore, the SNIAFL approach, which is to based on the idea of combining IR and static analysis, seems to be quite effective in addressing the feature location problem according to current experimental results. However, some abnormal cases raised in the experiments indicate that our approach may still need further improvements.

In the future, we will focus on doing experiments on larger software systems to further evaluate the feasibility and usability of our approach. To apply IR more effectively and ease its automation, we will exploit more sophisticated preprocessing mechanisms to deal with identifiers using abbreviations and/or acronyms. We will also look at possible ways to reduce the number of pseudoexecution traces generated by our approach. Inspired by the retrospection used in the pruning of decision trees, we will investigate whether this idea can boost the pruning step in our approach. Due to the imprecise nature of BRCG, we are also planning to apply other static representations and/or static code analysis techniques (such as program slicing [Weiser 1984]) for this purpose.

ACKNOWLEDGMENTS

We appreciate the timely help of Ms. Dan Hao and Mr. Hao Zhong, who performed part of the experimental study. We are also grateful to the anonymous referees for their helpful suggestions.

REFERENCES

ANQUETIL, N. AND LETHBRIDGE, T. 1998. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto, April). IEEE Computer Society, Washington, D.C. 84–93.

ANTONIOL, G., CANFORA, G., CASAZZA, G., AND DELUCIA, A. 2000. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the 16th International Conference on Software Maintenance*. (San Jose, Calif. Oct.). IEEE Computer Society, Washington, D.C. 40–49.

ANTONIOL, G., CANFORA, G., CASAZZA, G., DELUCIA, A., AND MERLO, E. 2002. Recovering traceability links between code and documentation. *IEEE Trans. Soft. Eng. 28*, 10, 970–983.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. ACM Press. New York.

BIGGERSTAFF, T., MITBANDER, B., AND WEBSTER, D. 1993. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering* (Baltimore, Md., May.). IEEE Computer Society, Los Alamitos, Calif. 482–498.

BOHNER, S. A. AND ARNOLD, R. S. 1996. *Software Change Impact Analysis*. IEEE Computer Society, Los Alamitos, Calif.

BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A., AND STONE, C. J. 1984. *Classification and Regression Trees*. Wadsworth, Belmont, Calif.

CHEN, A., CHOU, E., WONG, J., YAO, A. Y., ZHANG, Q., ZHANG, S., AND MICHAIL, A. 2001. CVS Search: Searching through source code using CVS comments. In *Proceedings of the 17th International Conference on Software Maintenance* (Florence, Nov.). IEEE Computer Society, Washington, D.C. 364–373.

CHEN, K. AND RAJLICH, V. 2000. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension* (Limerick, Ireland, June). IEEE Computer Society, Washington, D.C. 241–249.

CORBI, T. A. 1989. Program understanding: Challenge for the 1990's. *IBM Syst. J. 28*, 2, 294–306.

CUBRANIC, D. AND MURPHY, G. C. 2003. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oreg., May). IEEE Computer Society, Washington, D.C. 408–418.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. 1990. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci. 41*, 391–407.

EICK, S., STEFFEN, J., AND SUMMER, E. 1992. Seesoft—A tool for visualizing line-oriented software statistics. *IEEE Trans. Softw. Eng. 18*, 11, 957–968.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2001. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the 17th International Conference on Software Maintenance* (Florence, Nov.). IEEE Computer Society, Washington, D.C. 602–611.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. *IEEE Trans. Softw. Eng. 29*, 3, 210–224.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. 1999. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, Boston, Mass.

GNU DC. An Arbitrary Precision Calculator. http://www.gnu.org/directory/GNU/bc.html.

GNU UNRTF. Converts from RTF to Other Formats. http://www.gnu.org/software/unrtf/unrtf.html.

GRISWOLD, W. G., YUAN, J. J., AND KATO, Y. 2001. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ont. May). IEEE Computer Society, Washington, D.C. 265–274.

GUPTA, N., MATHUR, A. P., AND SOFFA, M. L. 1998. Automated test data generation using an interactive relaxation method. In *Proceedings of the 6th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Lake Buena Vista, Fla., Nov.). ACM Press, New York, 231–244.

LI, X.-B., SWEIGART, J., TENG, J., DONOHUE, J., AND THOMBS, L. 2001. A dynamic programming based pruning method for decision trees. *J. Comput. 13*, 4, 332–344.

LICATA, D. R., HARRIS, C. D., AND KRISHNAMURTHI, S. 2003. The feature signatures of evolving programs. In *Proceedings of the 18th International Conference on Automated Software Engineering* (Montreal, Que., Oct.). IEEE Computer Society, Washington, D.C. 281–285.

MALETIC, J. I. AND MARCUS, A. 2001. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ont., May). IEEE Computer Society, Washington, D.C. 103–112.

MARCUS, A. AND MALETIC, J. I. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oreg., May). IEEE Computer Society, Washington, D.C. 125–135.

MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. I. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering* (Delft, Nov.). IEEE Computer Society, Washington, D.C. 214–223.

QIN, T., ZHANG, L., ZHOU, Z., HAO, D., AND SUN, J. 2003. Discovering use cases from source code using the branch-reserving call graph. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference* (Chiang Mai, Thailand, Dec.). IEEE Computer Society, Washington, D.C. 60–67.

RAJLICH, V. AND WILDE, N. 2002. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension* (Paris, June). IEEE Computer Society, Washington, D.C. 271–278.

ROBILLARD, M. P. AND MURPHY, G. C. 2002. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Fla., May). IEEE Computer Society, Washington, D.C. 406–416.

SALTON, G. AND LESK, M. E. 1968. Computer evaluation of indexing and text processing. *J. ACM 15*, 1, 8–36.

SALTON, G. 1971. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice Hall, Englewood Cliffs, N.J.

SHAN, J., WANG, J., AND QI, Z. 2001. On path-wise automatic generation of test data for both white-box and black-box testing. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference* (Macau, China, Dec.). IEEE Computer Society, Washington, D.C. 237–240.

TIP, F., KIEZUN, A., AND BAEUMER, D. 2003. Refactoring for generalization using type constraints. In *Proceedings of the 18th Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, Calif. Nov.). ACM Press, New York, 13–26.

TURVER, R. J. AND MALCOLM, M. 1994. An early impact analysis technique for software maintenance. *J. Softw. Maintenance: Res. and Pract. 6*, 1, 35–52.

WEISER, M. 1984. Program slicing. *IEEE Trans. Soft. Eng. 10*, 4, 352–357.

WILDE, N., GOMEZ, J. A., GUST, T., AND STRASBURG, D. 1992. Locating user functionality in old code. In *Proceedings of the 8th International Conference on Software Maintenance* (Orlando, Fla., Nov.). IEEE Computer Society, Washington, D.C. 200–205.

WILDE, N. AND SCULLY, M. C. 1995. Software reconnaissance: Mapping program features to code. *J. Softw. Maintenance: Res. Pract. 7*, 1, 49–62.

WILDE, N., BUCKELLEW, M., PAGE, H., RAJLICH, V., AND POUNDS, L. 2003. A comparison of methods for locating features in legacy software. *J. Syst. Softw. 65*, 2, 105–114.

WONG, W. E., GOKHALE, S. S., HORGAN, J. R., AND TRIVEDI, K. S. 1999. Locating program features using execution slices. In *Proceedings of the 2nd Symposium on Application-Specific Systems and Software Engineering Technology* (Richardson, Tex., Mar.). IEEE Computer Society, Washington, D.C. 194–203.

YAU, S. S., NICHOL, R. A., TSAI, J. J., AND LIU, S. 1988. An integrated life-cycle model for software maintenance. *IEEE Trans. Softw. Eng. 15*, 7, 58–95.

ZHAO, W., ZHANG, L., LIU, Y., LUO, J., AND SUN, J. 2003. Understanding how the requirements are implemented in source code. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference* (Chiang Mai, Thailand, Dec.). IEEE Computer Society, Washington, D.C. 68–77.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J., AND YANG, F. 2004. SNIAFL: Towards a static noninteractive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering* (Edinburgh, UK, May). IEEE Computer Society, Washington, D.C. 293–303.

Received November 2004; revised July 2005; accepted December 2005