PAPER

Feature Location in Source Code by Trace-Based Impact Analysis and Information Retrieval

Zhengong CAI[†], ***Student Member*****, Xiaohu YANG**[†a)]**, Xinyu WANG**[†], ***and*** **Aleksander J. KAVS**[††], ***Nonmembers***

SUMMARY Feature location is to identify source code that implements a given feature. It is essential for software maintenance and evolution. A large amount of research, including static analysis, dynamic analysis and the hybrid approaches, has been done on the feature location problems. The existing approaches either need plenty of scenarios or rely on domain experts heavily. This paper proposes a new approach to locate functional feature in source code by combining the change impact analysis and information retrieval. In this approach, the source code is instrumented and executed using a single scenario to obtain the execution trace. The execution trace is extended according to the control flow to cover all the potentially relevant classes. The classes are ranked by trace-based impact analysis and information retrieval. The ranking analysis takes advantages of the semantics and structural characteristics of source code. The identified results are of higher precision than the individual approaches. Finally, two open source cases have been studied and the efficiency of the proposed approach is verified.

key words: *feature location, impact analysis, class ranking, information retrieval, trace extension*

1. Introduction

Feature location in source code is essential for software maintenance and evolution. It is nearly impossible to fix a bug or enhance a given requirement before locating the source code of interest. However, the complete comprehension of a system without documents is so time consuming that the software maintainers can not always understand the whole system, especially for the large-scale system, before alternating the source code. Thus, the as-needed comprehension - locating feature in the source code - is preferable for program comprehension and system maintenance [22].

The feature may be a functional or non-functional requirement of a system. In fact, it is not easy to locate non-functional feature in source code. Most of the existing research has focused on the functional feature location. A large amount of feature location approaches have been proposed to find the source code of most interest in the last decade [16]. The approaches can be categorized into static analysis, dynamic tracing and the hybrid ones. The static ones include information retrieval and dependency analysis, but both of them may involve irrelevant code since they do not execute the source code during the location process. The dynamic trace is an interactive approach and is considered to obtain the most precise results. However, the dynamic analysis needs plenty of test cases for correct results. Combining the static and dynamic analysis could avoid the defeats of a single approach to achieve better results [20].

This paper proposes a new combined approach for the feature location in source code by the trace-based impact analysis and information retrieval. The approach is divided into two phases. First, the source code is traced by dynamic analysis and static inference. In this phase, both the source code and the feature are preprocessed. The source code is instrumented and traced by executing the selected test cases. The execution trace is extended by analyzing its control flow, i.e. the branches related to the execution trace. Then, the classes in the extended execution trace are ranked by the trace-based impact analysis and information retrieval. The corpus is built from the traced source code and the description of the feature is used as a query for semantic analysis by filtering the stop words. The information retrieval measures the semantic similarity between the traced source code and the feature description. The trace-based impact analysis evaluates the static dependency between the trace and the other classes.

This approach provides a combined ranking strategy for the irrelevant classes filtering which is a common problem in the location approaches. It takes advantages of static dependency analysis, information retrieval and dynamic trace. Some combined location approaches have been proposed but their ranking metrics are still weak to find closely related code or filter the irrelevant classes [11]. Our approach has three advantages:

- It is applicable for analyzing unfamiliar code since only one primary scenario is needed. The potentially relevant classes are considered by extending the execution trace. However, the traditional dynamic analysis needs plenty of scenarios with or without the feature to locate the relevant source code.
- The classes in the extended trace are ranked by information retrieval and impact analysis. The intrinsic challenges of them like synonymous can be controlled. The ranking aids on filtering irrelevant classes brought by static inference.
- The trace is introduced into impact analysis for refinement. The refined impact analysis would improve class dependency metrics.

Finally, two open source cases are analyzed to evaluate

Manuscript received March 9, 2011.
Manuscript revised August 21, 2011.

[†]The authors are with the Computer College, Zhejiang University, Hangzhou, Zhejiang, 310027, China.
[††]The author is with the State Street Corporation, Boston MA 02110–0351, U.S.A.
a) E-mail: yangxh@zju.edu.cn

DOI: 10.1587/transinf.E95.D.205

our approach. Some features of the two systems are selected for evaluation. The analysis results are discussed and compared with those of the individual approaches. Our proposed approach is approved to be of more efficiency with high precision from the experiment results. Also, some parameters in our approaches are discussed. The remaining of this paper is organized as follows. Section 2 gives the prior work related to our research. Section 3 introduces our proposed approach and its critical phases. In Sect. 4, two open source cases are analyzed to evaluate the approach and the experiment results are discussed. Section 5 concludes our work and discusses the future work.

2. Related Work

A large amount of research has been done on locating features in the source code. These researches can be categorized into three groups: static techniques, dynamic analysis and the hybrid ones. All the three types of approaches are discussed as follows.

The static techniques for feature location include the pattern matching [1], IR-based [2], [17] and dependency graph [5]. Pattern matching [1] like grep is a query-based approach to find source code with specific terms. This approach requires the analyst's prior knowledge on the system and the problem domain. For unfamiliar system, it's nearly impossible for the analysts to design good queries. Usually, many queries are needed before correctly locating the target source code. The IR-based approach [2] is similar as the grep-based approach. In this approach, latent semantic indexing (LSI) is adopted to measure the similarities between the queries and the source code. Different from the grep-based approach, a list of ranked source code is returned in descending order, which would provide useful knowledge for further comprehension. The only way to find the code of interest is looking through the returned results. Both grep-based and IR-based approaches are based on the hypothesis that the identifiers and comments in the source code are meaningful [3], [23]. In addition, the structure of the code is not considered in this approach. Some strategies were introduced for improving the LSI-based approach by considering the characteristics of the source code [4]. The dependency analysis is one of the structural analysis techniques. The analyst needs to navigate the Abstract System Dependency Graphs (ASDG) [5] to find relevant implementations. Any mistake would lead to backtrack of the dependency analysis. Although the approach does not need the related domain knowledge, an entrance or key point for the feature is required. The entrance identification also needs the prior comprehension of the source code. The combination of these static techniques is also studied in the last years. Zhao et al [8] provides a feature location approach by combining the information retrieval and branch-reserving call graph. This approach first identifies the initial connections between features and the computational units by information retrieval. Then a BRCG - branch-reserving call graph - is applied to further recover the specific computational units for the feature. The approach claims no interactive is needed. However, all the static analysis approaches rely heavily on analysts' knowledge and ignore actual execution traces, which may take some irrelevant source code.

Software reconnaissance [6] is a dynamic feature location approach. Both the cases with and without this feature are executed to locate feature-relevant classes. Only the source code existing in the execution traces is considered as relevant to the feature. The approach has been extended by introducing rankings to measure how closely the source code relevant to the feature. Dynamic feature location also has some disadvantages [7] like requiring plenty of test cases. In [10], the scenarios are executed incrementally to construct a scenario-feature and scenario-code mapping matrix. For each item in this matrix, 1 for the code is invoked to implement the scenario, whereas 0 for not. The matrix is converted into a sparse formal concept lattice, in which scenario is selected as attribute and code as object. The code is classified into specific, co-specific, shared, relevant and irrelevant. In this way, the results are comprehensible for analysts. It is claimed that the approach can be applicable for unfamiliar source code. However, the results by dynamic analysis are heavily affected by the quality of test cases. Prior knowledge is also needed for analysts to design good test cases. To locate a feature, over one scenario should be executed to verify whether a function is specific to the feature or shared with other features.

To avoid the defeats of both static and dynamic approaches, some researchers tried to combine them [9], [11]. In [11], the ranking based on both dynamic probabilistic and information retrieval is calculated to measure the contribution of a computational unit to the specific feature. More than one feature should usually be considered when locating one feature. However, the dynamic analysis in this approach also needs plenty of test cases, especially for calculating the probabilistic value. In [9], the dynamic analysis is executed to obtain a set of computational units by tracing. The impact of the computational unit is analyzed to measure the contribution of the computational unit to the feature. One new approach [21] is proposed using the execution trace of only one single scenario. This approach executes one single scenario, and then the computational units in the trace are ranked using information retrieval method. The ranking of all the executed computational units is based on the textual similarity to the query. However, the feature trace using a single scenario may miss some relevant source code. The static inference is also used to extend the feature trace [20], but no practical idea to filtering the irrelevant code brought by static inference.

Additionally, some other approaches were also proposed for specific system models, like for mainframe [14], distributed system [12], SOA [13], etc. However, they are limited by specific applications. Therefore, we propose a new approach by a hybrid ranking strategy for the feature trace, in order to avoid plenty of test cases and heavily reliance on prior knowledge.

3. Combined Feature Location Approach

This paper proposes a new combined approach for locating features in source code. It takes advantage of dynamic tracing, static inference and text similarity, as in Fig. 1. The approach is not only to find the classes implementing the feature, but also evaluate the connectivity strength of a class to the feature. First, the instrumented system is executed to locate which classes implement the feature. The dynamic execution trace is extended by control flow analysis to cover more classes that are potentially relevant to the feature but missed in dynamic tracing. Then, each class in the trace is ranked by hybrid ranking analysis which combines trace-based impact analysis and semantic analysis. The classes in the obtained trace are converted into a set of documents for further semantic searching, each class as a document after pruning the operators and key words related to specific programming language. The class ranking is not only to measure the connectivity strength of each class to the given feature, but also to filter the irrelevant classes that brought by the extension of static inference.

3.1 Feature Trace and Extension

The relationships between a class and the given feature can be categorized into three types: *specific*, *relevant* and *irrelevant*. The feature trace in this approach is to find all the classes that implement this feature, including the specific and relevant ones. A class c implements the feature f can be represented as $M(c, f)$. Specific means a class is executed only for that feature, represented as $S(f)$. It is a strong relevant relationship.

$$S(f) = \{c|M(c, f) \wedge (\forall f' \neq f, \neg M(c, f'))\} \tag{1}$$

If the class also serves other features, it is relevant to but not specific for that feature, represented as $R(f)$.

$$R(f) = \{c|M(c, f) \wedge (\exists f' \neq f, M(c, f'))\} \tag{2}$$

If one class contributes nothing to that feature, it is called irrelevant to the feature, represented as $I(f)$.

$$I(f) = \{c|\neg M(c, f)\} \tag{3}$$

In the formulas, $C(f) = S(f) \cup R(f)$ represents all the classes that implement the feature f, including both specific and relevant relationships. Thus, $C(f)$ is also the class set that will be located by feature trace.

In our approach, we propose an extended tracing algorithm to cover all the potentially relevant classes by analyzing the control flow in the execution trace. First, a single scenario is executed to scale the execution trace corresponding to the software feature under study. The source code is instrumented to record the classes and their methods that are executed. The trace result is a set of classes - exactly, a sequential set of the executed methods, including the branch statements. Then, the execution trace is extended to cover more potential classes by analyzing the control flow. The classes in the execution trace are indeed related to the studied feature. However, some classes may be missed if the source code is traced by only a single scenario, especially for a complex feature. On the other hand, it is too difficult for the analysts without prior knowledge to design such a perfect scenario that can cover all the potential classes for the feature. Thus, the static workflow analysis is introduced to extend the single execution trace to find more potential feature-relevant classes. The extension focuses on the control flow, i.e. branch statement, as in the algorithm 1.

Algorithm 1: Execution Trace Extension

```
Input: entrance - the entrance statement of an execution trace
trace - trace results
sc - system source code
Output: c - extended class set
 1: c=c ∪ {$classoftheentrance$};
 2: currentStatement = entrance;
 3: while currentStatement do
 4:     if currentStatement is branch then
 5:         $getthebranchesBexcepttheexecutedone$;
 6:         for all branch entrance $be \in B$ do
 7:             $c=c \cup ExecutionTraceExtension(be, trace, sc)$ ;
 8:         end for
 9:     end if
10:     if currentStatement is function then
11:         traverse the function invocation recursively;
12:         add the owners of all the invoked methods and those
            impacting them into c;
13:     end if
14:     currentStatement = currentStatement.successor;
15: end while
16: return c;
```

In line 9, the function statement is traversed to obtain the invoked methods using the static dependency analysis. The invoked methods are added as the extension trace. Also, the methods that the invoked methods depend on are considered as trace extension. The analyzed dependencies include inheritance, composition and use. The owner classes of these invoked methods or the methods that the invoked methods depend on are added to the class set c. In fact, the invoked method may be abstract or overridden. All the concrete implementations of the abstract methods or the classes overriding the invoked methods would also be added into the class set c by static dependency analysis, if their actual execution can not be located by manual analysis.

All classes in the extended execution trace are considered as the traced results. However, the extension by static inference may bring some irrelevant code. The ranking analysis of the traced results is used to filter the irrelevant source code by traced-based impact analysis and semantic analysis. Also, the ranking is for measuring the connectivity strength of the classes to the feature. The engineers only need to focus on the classes with high ranking values when understanding or maintaining a feature. For example, when modifying a feature, the specific classes can be changed without

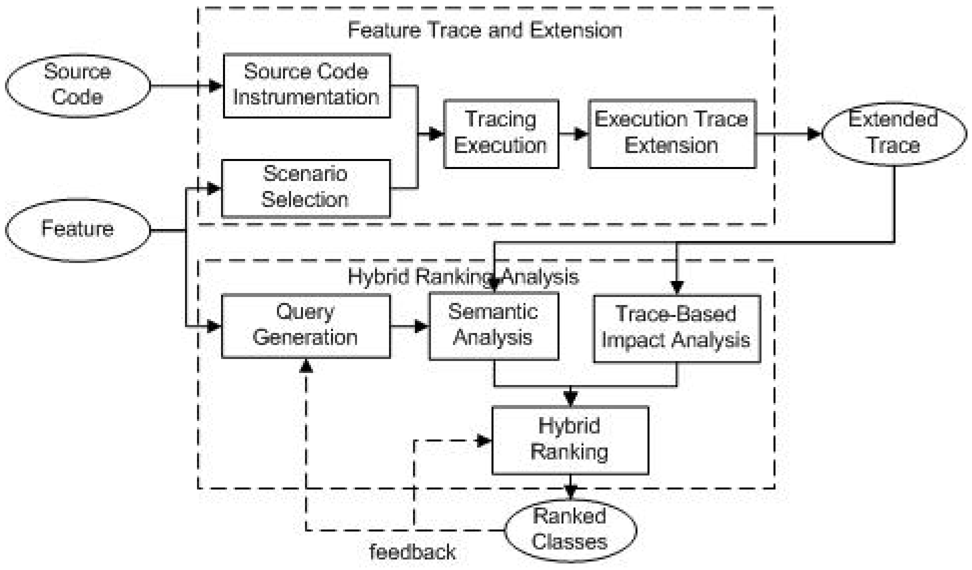

Fig. 1 The feature location process using the combined approach.

considering the impact on other features, while other features should be evaluated if a relevant classes needs to be modified.

3.2 Hybrid Ranking Analysis

The classes in the extended trace contain the specific, relevant and even several irrelevant classes of the feature. The contribution of each class to feature implementation should be measured in a unified way to locate the closely relevant classes. A metric combining both structural and semantic analysis is proposed to rank the classes. In the following subsections, we first introduce the trace-based impact analysis and semantic analysis separately, and then give the hybrid ranking approach.

3.2.1 Trace-Based Impact Analysis

Change impact analysis is one of the well-known ways to analyze the coupling between one class and the other classes in the system. Two classes are coupled if one class is affected when the other class changes. As discussed in the related work, some researchers have introduced change impact analysis to solve the feature location problem. The impact analysis of one class on the others is used to locate features in source code [15]. In this paper, a low impact value means the class is closely relevant to the feature. However, it is meaningless if its impacted classes belong to the execution trace of that feature. In other words, even if one class affects many classes, the relevance can also be high if most of the impacted classes are also in the feature trace. Thus, a trace-based impact analysis is proposed to measure how closely a class belongs to the execution trace. In the trace-based impact analysis, the impact of a class in the trace is analyzed by the directed class dependency graph, where the dependency between the class and the feature trace is included as well as the dependency between the class and the other classes not in the feature trace.

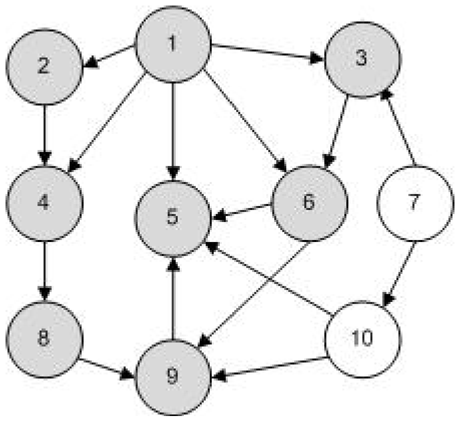

Fig. 2 A directed class dependency graph example.

In the directed class dependency graph, each class is treated as a single node, and the directed edge means one class depends on the other (pointed to). The direction from *A* to *B* means the implementation of class *A* depends on class *B*, i.e. the change of *B* affects *A*. The direction of three types of dependency is: inheritance is from child to parent, composition is from whole to part, and invocation is from caller to the called. The directed class dependency graph of the whole target system is constructed by static analysis. The classes in the feature trace are marked in shadow, as in Fig. 2. It gives a directed class dependency graph with feature trace.

Different from the traditional impact analysis [15], the trace-based impact analysis provides a positive value to measure the impact of a class on the trace. The higher the value, the more impact of the class is to the feature trace. In this way, the trace-based impact analysis can be easily combined with other ranking results like those by information retrieval. The impact strength of a class c to the given trace is measured as follows.

$$R_D(c) = \begin{cases} e^{\frac{|EC(c)|}{|CS|}-1} & |\mathrm{AC(c)}|=0 \\ \frac{|AC_i(c)|}{|AC(c)|} * e^{\frac{|EC(c)|}{|CS|}-1} & \text{otherwise} \end{cases} \tag{4}$$

CS: the total class set of the system.

$AC(c)$: the set of the classes directly or indirectly affected when class c is modified.

$EC(c)$: the set of classes that directly or indirectly affects the class c.

$AC_i(c)$: the set of the classes directly or indirectly affected in the execution trace i when class c is modified.

In Eq. (4), $\frac{|AC_i(c)|}{|AC(c)|}$ measures the strength that the class c affects on the trace compared with all the classes it affects. The more classes affected by c in the trace, and the less classes that c affects in total, the larger the impact strength is. The value is 0 if it has no affects on the classes in the trace but affects other classes. The class is also treated as irrelevant to the feature (i.e. in *I(f)* of feature *f*). On the other hand, the value 1 means all the affected classes in the trace, i.e. the class is in *S(f)*. $e^{\frac{|EC(c)|}{|CS|}-1}$ is a revision parameter based on the efferent class number. The more classes that affect this class, the more probably it is important in the feature trace. The upper limit value 1 can be achieved only when all the classes in the system affect class c. Thus, the impact value belongs to [0,1], i.e. $R_D(c) \in [0, 1]$.

Take the directed class dependency graph in Fig. 2 as an example, the shadowed nodes 1-6, 8 and 9 are treated in the execution trace, whereas the classes 7 and 10 are not. Not all of the classes traced are specific for that feature since some of them may also contribute to the classes not in the trace, e.g. class 3, 5, 6 and 9. Thus, the impact of each class to a given feature should be calculated by considering its supporting on other features using Eq. (4). Table 1 shows the ranking values by trace-based impact analysis. From the results, we find that both class 6 and 8 are affected by two other classes and they affect three classes separately. All the classes affected by class 8 are in the extended execution traces, but some of those by class 6 are not. Thus, their connectivity to the given feature is different and can be represented by the ranking value.

3.2.2 Semantic Analysis by VSM

The structural dependency between classes relies on the system architecture design or programming styles of the developers. In Table 1, the impact strength of class 5 is higher than that of class 3. In fact, class 3 may implement some business function of the feature since other three classes support it. However, class 5 may be a utility class since it has a high fan-in but no fan-out. The difference can not be identified only using the trace-based impact analysis. The semantic similarity can be applied as a complement. With good programming style, the identifiers and comments in the source code contain some potential knowledge for the implemented feature. Comprehending these identifiers could facilitate the process to find the target classes. Semantic analysis has become one of the popular ways to find the relationships between source code and domain knowledge [2], [17].

Table 1 Trace-based impact analysis results of Fig. 2.

j	$AC(j)$	$AC_i(j)$	$\lvert EC(j)\rvert$	$R_D(j)$
1	0	0	7	0.741
2	1	1	4	0.549
4	2	2	3	0.497
8	3	3	2	0.449
9	8	6	1	0.306
6	3	2	2	0.300
5	9	6	0	0.286
3	2	1	3	0.248

The identifiers in the well-programming source code represent the business knowledge that it implements. The text similarity between source code and feature reflects the links from feature to its implementation. Vector Space Model (VSM) is an information retrieval way to measure the text similarity between source code identifiers and feature description [2]. In our approach, each class in the execution trace is converted into a document. The document set is represented as $D = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n\}$, where $\mathbf{d}_i = \{t_{1i}, t_{2i}, \ldots, t_{ni}\}$ .The identifiers are selected as the terms of the source code. They can be classified into four groups: class identifiers, method/field identifiers, parameter identifiers and comment identifiers. A weight is assigned for each type of identifiers, inspired from the previous research like [23]. Then, a set of terms are extracted from the given feature description, represented as $\mathbf{T} = \{t_1, t_2, \ldots, t_m\}$. These terms are used as queries to find similar source documents from the feature trace. The semantic similarity between the query - i.e. feature description - and a document is defined using the cosine value of the vectors according to the query and the source code.

$$R_S(\mathbf{d}_j) = Sim(\mathbf{T}, \mathbf{d}_j) = \frac{\sum_{i=1}^{N} w_{ij} * w_{it}}{\sqrt{\sum_{i=1}^{N} w_{ij}^2} * \sqrt{\sum_{i=1}^{N} w_{it}^2}} \tag{5}$$

where,

$N = |\mathbf{T} \cup \mathbf{d}_j|$: the total number of items.

w_{it}: the term frequency in the query $\mathbf{T}$.

$w_{ij} = tf_i * w_T$: tf_i is the term frequency in $\mathbf{d}_j$ and $w_T \in [0, 1]$ is the weight of the identifier type. The weights are: *class¿method/field¿parameter¿comments*.

The text similarity between the document and query is used as the connectivity of the class to the given feature. Its value belongs to [0,1]. 0 means the class is irrelevant to the feature, while 1 means identical to the feature description. The semantic analysis is used not only to measure the class ranking, but also to filter the semantic irrelevant classes introduced by trace extension.

3.2.3 Class Ranking by Hybrid Analysis

Analyzing the classes by trace-based impact analysis is hypothetically treated all the traced classes as relevant to the feature. The semantic analysis indeed considers the text similarity between source document and feature description. However, the semantic analysis has some intrinsic disadvantages for source code analysis [3]. As discussed above, both the trace-based impact analysis and semantic analysis provide positive measurement of the class ranking from different views. Trace-based impact analysis measures the impact strength between one class and other classes. The higher impact of one class on the others in the trace, the more it contributes to the given feature. On the other hand, the semantic analysis measures the text similarity between the feature description and class documents. This is not considered in structure-based analysis but is also very helpful. Thus, in our approach, the two static analysis approaches for class ranking are combined. The classes with high ranking values are considered as relevant to the given feature. $R_D(c)$ and $R_S(c)$ measure the relevance between class c and the given feature from structural and semantic views separately. The two metrics are combined by a parameter ∂ to integrate structural and semantic analysis, as in Eq. (6).

$$R(c) = \partial * R_D(c) + (1 - \partial) * R_S(c) \tag{6}$$

$\partial \in [0, 1]$ is used to balance the contribution of the trace-based impact analysis and the semantic analysis. $\partial = 0$ means only the semantic metric is considered, if the feature is well documented, whereas $\partial = 1$ means only the trace-based impact analysis is considered if lack of feature description. For other ∂ values, both of the ranking analysis metrics are adopted. $R(c)$ measures the hybrid ranking value of class c to the given feature.

The hybrid ranking in algorithm 2 is an interactive process described in the following algorithm. In the algorithm, the initial query and parameter are set for ranking analysis of each class in the trace. The statements 5 and 6 are for filtering the irrelevant classes that are brought by trace extension. If the classes extended by static inference have no semantic similarity with the feature, it can be treated as irrelevant. From statement 13, the classes with top-k ranking values are selected and evaluated. If the evaluation result is not satisfied (the top-k classes are accepted by engineers), the ∂ is updated using an incremental step and the hybrid ranking is re-executed. The value k can be determined by analyzing the largest ranking value gap of the neighboring classes in the ranking list. The classes above this gap are treated as top-k classes. It makes sense since other parts of the feature could be located by analyzing the top-k classes.

The k classes with high ranking values are selected as feature-relevant classes by locating largest ranking value gap. The iterative process provides a set of top-k classes, which aids the engineers to identify best results by comparison. It would eliminate the requirements of extra knowledge for top-k selection. The appropriate results can be obtained by recursive adjustment of ∂ and $\delta\partial$. It would greatly decrease the iteration number. E.g. $\delta\partial = 0.1$ for ∂, then $\delta\partial = 0.01$ for a smaller scale 0.1.

Algorithm 2: Hybrid Ranking

```
Input: trace - extended trace class set;
q - query;
$\partial$ - the parameter for combination;
$\triangle\partial$ - an adjustment value for $\partial$;
Output: $V_r$ - ranking values of the trace;
 1: while $\partial \leq 1$ do
 2:     $V_r = null$;
 3:     for all c in the trace do
 4:         $vs(c)$ =semantic ranking value by Eq. (5);
 5:         if $v(c) == 0$ and c is obtained by static extension then
 6:             $v(c) = 0$;
 7:         else
 8:             $vd(c)$ =impact ranking value by Eq. (4);
 9:             $v(c) = \partial * vd(c) + (1 - \partial) * vs(c)$;
10:         end if
11:         insert $(c, v(c))$ into $V_r$;
12:     end for
13:     select top-k class set V with largest gap with others;
14:     if V not satisfied then
15:         $\partial = \partial + \triangle\partial$;
16:     end if
17: end while
18: return $V_r$;
```

4. Experiments and Analysis

Our proposed feature location approach is evaluated by analyzing two open source cases: *serp* - a library for bytecode code analysis, and *jhotdraw* (version 7.5.1) - a GUI-based tool for graphic draw. One critical feature is selected for *serp* and five features are for *jhotdraw*. The precision is discussed by analyzing the "correct" classes in located results. Also, the selection of parameter for combining the two analysis approaches is discussed for better combination. Finally, the threats affecting this experiment are discussed.

4.1 Experiment Process

Not all the phases of the experiment process are implemented from scratch. Some open source tools or techniques are used to assist our experiment. The open source eclipse-based toolkit BIPTK (Bytecode Instrumentation Profiling and Toolkit for Java) is used to instrument the bytecode [18]. The execution trace is obtained by running the instrumented target system and is extended according to our extension algorithm. Then the classes in the extended trace are ranked according to the hybrid impact and semantic analysis, as in Fig. 3, where the rectangles represent the analysis actions and the ellipses represent the intermediate results.

In these experiments, the weight w_T in Eq. (5) is set to 0.9, 0.8, 0.7 and 0.6 for the identifiers of *class*, *method/field*, *parameter* and *comments*. The parameters ∂ and $\triangle\partial$ are set to 1 and 0.1 in all these experiments.

4.2 Cases and Results

The first objective case *serp* is an open source library for

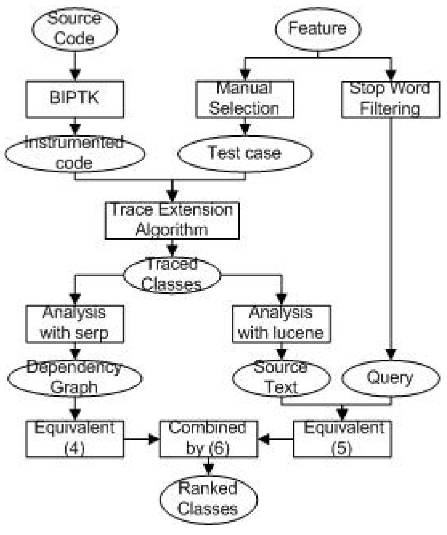

Fig. 3 The experiment process and its critical techniques.

analyzing the java bytecode. This is also the library that is used in the experiment for directed class dependency analysis. The library has 99 classes and provides some supporting functions for java byte code analysis. One supporting function is selected as the functional features for further feature location. The authors are familiar with the application of this case, so the results validation is more trustable by using source description as query.

The function *loading byte code classes* is one of the critical features of the library. To locate the related code that implementing this feature, two types of inputs are considered: test cases and query text. For the test case, we select a set of .class files for loading. For the query text, the description of the function is selected (http://serp.sourceforge.net/site/apidocs/serp/bytecode/Project .html#loadClass(serp.bytecode.BCClass)) - "*Load the bytecode from the given class file. If this project already contains the class in the given file, it will be returned. Otherwise a new BCClass will be created from the given bytecode*". To balance the effect of the trace-based impact analysis and semantic analysis, we set the parameter ∂ to 0.5. The ranking values in descending order are given in Table 2. The ranked classes are in the second column. High ranking values were assigned to the classes implementing the critical functions, which can be validated by the source code analysis manually.

From the ranking results, we select the top 4 classes which have the largest gap with others. In the four classes, *Project* is the entry for bytecode class loading and the other three for loading and management of loaded classes. They are the critical parts for class loading and other related classes can be reached by dependency analysis. This is confirmed by one co-author who is familiar with the package.

Besides analyzing the open source library *serp*, another open source program - *jhotdraw* [19]- is also discussed. For this case, we selected some low-granularity features, such as the GUI operations. The classes implementing these operations are located by our approach. The

Table 2 Ranking results of serp.

No.	Class Name	$R(c)$
1	serp.bytecode.Project	0.6314
2	serp.bytecode.NameCache	0.5425
3	serp.bytecode.BCClassLoader	0.4750
4	serp.bytecode.BCClass	0.4057
5	serp.byptecode.ConstantInstruction	0.2607
6	serp.byptecode.LocalVariableInstruction	0.2504
7	serp.bytecode.SourceFile	0.2433
8	serp.byptecode.Entry	0.1889
9	serp.bytecode.Exception	0.1518
10	serp.bytecode.ObjectState	0.1504
11	serp.bytecode.BCMember	0.0941

dynamic action description for dynamic tracing and package comments/description as query of information retrieval are listed in Table 3. For query generation, the package with similar terms is selected and the package description is converted to a query manually. The ∂ for best results will be discussed in the next section. To improve the efficiency, the trace is started when executing the operation to filter system initialization. The benefit of this partial trace is that the tracing result set for system initialization is very large. Filtering these would greatly improve the efficiency.

The ranking results for the five features in Table 3 are given in Table 4. The class number in the execution trace is given in the column |Traced Classes|. The number of classes extended is in the column |Extended Classes|, not including the traced classes. The class number by using VSM in the trace including extended ones is given in |Classes by VSM|. |In Trace| and |In Extension| are for classes with similarity higher than 0 in execution trace and its extension. The ranking result with highest precision (discussed in Sect. 4.3) is in |Ranked Results|. In the ranked results, the correct ones are given in |Correct Results|. Similarly, |In Trace| and |In Extesion| are for the results in execution trace and its extension.

In our experiment, for filtering the irrelevant classes introduced by extension, we ignore the classes in the extended trace but not in the execution trace if their semantic ranking value is 0. This action could decrease the effort of manually evaluation of the results. Additionally, the ranking analysis based on trace could greatly eliminate the irrelevant classes that may be involved by just VSM. The top-k classes are selected by analyzing the greatest gap between two neighboring classes in the ranking list. The classes above the gap are treated as top-k classes.

From Table 4, we can find that some correct results were located in trace extension but not traced. Take the feature 4 as an example, the top 7 classes were selected only using dynamic tracing (i.e. without extension). After extension, other three classes were located. They are "*org.jhotdraw.draw.tool.TextAreaCreationTool*", "*org.jhotdraw. gui.plaf.palette.PaletteLookAndFeel*", and "*org.jhotdraw.gui. JFontChooser*". All of them are relevant to the text format, but missed without trace extension. The results of both with and without extension are obtained

Table 3 The trace actions and query source of the features for *Jhotdraw*.

No.	Dynamic Trace Action Description	Query Source
1	Open the draw application GUI, click the rectangle button and draw a rectangle using the Bezier path	http://www.randelshofer.ch/oop/jhotdraw/javadoc751 /org/jhotdraw/draw/Drawing.html
2	Open the pert application, draw two tasks from the GUI and connect them by a dependency. The task has a name field, a duration field and a start time field.	http://www.randelshofer.ch/oop/jhotdraw/javadoc751 /org/jhotdraw/samples/pert/PertApplicationModel.html
3	Open the sample draw application and select to show the text attributes. Double click the item in the list to view all the font types. The types are listed in a family box with classifying, and also its sub catalogues are given.	http://www.randelshofer.ch/oop/jhotdraw/javadoc751 /org/jhotdraw/gui/fontchooser/package-summary.html
4	From an open draw application GUI, select the text button, draw a text area and insert a text in a draw application GUI. Then format the text to bold and red.	http://www.randelshofer.ch/oop/jhotdraw/javadoc751 /org/jhotdraw/draw/text/package-summary.html
5	From the open svg applet GUI, select the text button and insert a text box in the svg applet and format the text to bold, 48pt and Italic	http://www.randelshofer.ch/oop/jhotdraw/javadoc751 /org/jhotdraw/samples/svg/SVGApplet.html

Table 4 The intermediate results and ranking results for *jhotdraw*.

			|Classes by VSM|		|Ranked Results|		|Correct Results|	
Feature No.	|Traced Classes|	|Classes Extended|	|In Trace|	|In Extension|	|In Trace|	|In Extension|	|In Trace|	|In Extension|
1	86	34	68	28	5	1	5	1
2	106	42	43	16	4	2	4	1
3	115	54	76	49	11	4	11	4
4	94	30	61	30	7	3	6	3
5	107	52	87	47	6	2	5	2

at the best parameter.

4.3 Analysis and Discussion

The class ranking by trace-based impact analysis and semantic analysis is more flexible and can lead to better results than the traditional approaches. The parameter is discussed using the second case - the *jhotdraw* and its five features in Table 3. The location precision for these features is given in Fig. 4. For each feature, the highest precision value is obtained with $\partial \in [0.3, 0.8]$. By analyzing the features and located classes, it prefers a lower ∂ value - i.e. semantic analysis - to obtain the best results for the feature / source code with sufficient documents. On the other hand, for the legacy system without available document, the ∂ value should be larger. The precision of the location results is defined as:

$$precisoin = \frac{|correctclassesbyourapproach|}{|identifiedclassesbyourapproach|} \quad (7)$$

The identified classes are the top-k classes selected by locating the greatest ranking value gap. The identified class number of the five features at highest precision has been given in Table 4 (i.e. |Ranking Results|). For other parameter ∂, the identified result number (including correct and incorrect classes) may be different, as in Fig. 5. Whether a class in top-k set is correctly identified was determined by manual evaluation of engineers.

Besides the combining parameter, the advantages of hybrid ranking are also discussed by comparing it with single semantic analysis. Single impact analysis is not discussed here because it can not filter irrelevant classes

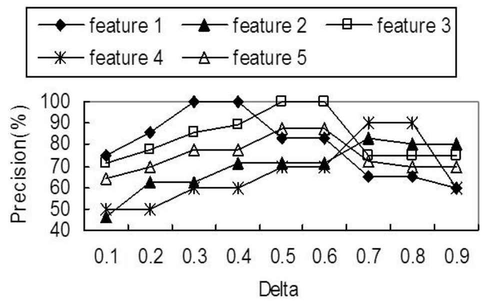

Fig. 4 The precision comparison as delta (∂) changes.

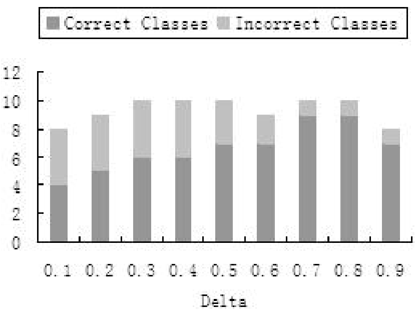

Fig. 5 The correct and incorrect classes in ranked results of Feature 4.

brought by trace extension. The ranking analysis is based on the extended execution trace instead of the whole system. Additionally, the selection of top-k classes were all based on locating the largest ranking value gap. As described in Sect. 3, our approach considers the characteristics of the source code. The identifiers for the classes and methods may represent more business terms than those in the comments. Thus, we discuss the improvement of the location precision by differentiating the identifiers, as in Fig. 6. The result precision of the five features in Table 3 for the *jhotdraw* has been analyzed. The precision by the refined information retrieval could improve about 10%. The location precision by our approach has about 5 20% improvement compared with that of the refined information retrieval.

In Fig. 6,

before - feature location by traditional semantic analysis, implemented by setting all the identifier weights w_{it} = 1.0 in VSM and setting $\partial = 0$.

refined - semantic analysis considering the characteristics of source code, implemented by setting $\partial = 0$.

our approach - the hybrid approach in this paper.

4.4 Threats to Validity

There already exist many techniques to locate feature in source code. Each feature location method has its characteristics and succeeds on the systems with special characteristics but none can fit all the systems. Thus, the hybrid approach for feature location is to overcome the defeats of the single location method. There are three inevitable issues that may affect the results of the experiment results. The issues may take some limitations to the experiment results. Thus, in this experiment, we tried to minimize the effect of the issues.

The first issue is the manual selection of the test cases. The ranking analysis is based on the tracing results. Any defeat in this step would affect the experiment results badly. Thus, the users familiar with the cases are invited to review our selected scenario.

The second issue is the query generation. The feature description from system document is used as the query to find classes with semantic similarity. Although this process needs little interaction, the description may not be complete and some terms in the description may be different from those in the source code. Thus, in the experiment, we preprocessed the synonyms manually.

The third issue is that the correct results for comparison are analyzed manually by the co-authors and other lab members. Thus, we use the analysis results confirmed by all the involved members as the correct results to minimize the potential experimental risks.

There are also some other issues, e.g. the boundary of the feature in as-needed trace. We selected the features that are triggered by external input in the experiment.

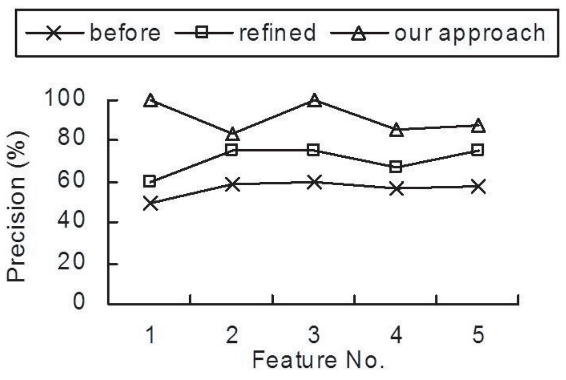

Fig. 6 The precision of different ranking techniques.

5. Conclusion

Feature location is an important step for program comprehension and software maintenance. To solve the feature location problem, we propose a new combined approach considering the trace-based impact analysis and semantic analysis. The trace-based impact analysis is a refined impact analysis metrics by considering the execution trace, and the semantic analysis considers the characteristics of the source code. Combining the two static analysis approaches could overcome the defeats from either of them. Two open source cases are analyzed and the results have been evaluated to approve its efficiency and precision. The contributions of our approach include:

- The location process only needs a single scenario selected by analysts. The potentially relevant code can be covered by introducing the trace extension based on workflow. Thus, it's applicable for unfamiliar code.
- The trace-based impact analysis and information retrieval are combined to avoid the defeats of either structural analysis or semantic analysis.
- Impact analysis is improved by introducing the trace.

Feature location is an ongoing research topic. A set of ranked classes are given by our approach but it still can not clearly distinguish the specific classes and common relevant classes. In the future, we will consider other techniques like formal concept analysis to aid our ranking metrics. In addition, more cases will be studied to discuss the parameters in our approach and the location of the compound feature will be studied.

References

[1] A. Marcus, V. Rajlich, J. Buchta, M. Petrnko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," Proc. 13th IEEE International Workshop on Program Comprehension, pp.33–42, 2005. [doi: 10.1109/WPC.2005.33]

[2] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," Proc. 11th Working Conference on Reverse Engineering, pp.214–223, 2004. [doi: 10.1109/WCRE.2004.10]

[3] D.P. Liu and S.C. Xu, "Challenges of using LSI for concept location," ACM SouthEast Conference, pp.449–454, 2007. [doi: 10.1145/1233341.1233422]

[4] G.Z. Lai, X.B. Wang, and C. Liu, "Analysis and improvement on retrieval methods for traceability links between source code and documentation," ACTA Electronica Sinica, vol.37, no.4A, pp.22–30, 2009. (in Chinese)

[5] K. Chen and V. Rajlich, "Case study of feature location using dependency graph," Proc. 8th International Workshop on Program Comprehension, pp.241–249, 2000. [doi: 10.1109/WPC.2000.852498]

[6] N. Wilde and M.C. Scully, "Software reconnaissance: Mapping program features to code," J. Software Maintenance: Research and Practice, vol.7, no.1, pp.49–62, 1995. [doi: 10.1002/smr.4360070105]

[7] A.D. Eisenberg and K.D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," Proc. 21st International Conference on Software Maintenance, pp.337–346, 2005. [doi: 10.1109/ICSM.2005.42]

[8] W. Zhao and L. Zhang, "SNIAFL: Towards a static non-interactive approach to feature location," ACM Trans. Softw. Eng. Methodol., vol.15, no.2. pp.195–226, 2006. [doi: 10.1145/1131421.1131424]

[9] A. Rohatgi, A.H. Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," Proc. 16th International Conference on Program Comprehension, pp.236–241, 2008. [doi: 10.1109/ICPC.2008.35]

[10] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," IEEE Trans. Softw. Eng., vol.29, no.3, pp.210–224, 2003. [doi: 10.1109/TSE.2003.1183929]

[11] D. Poshyvanyk, A. Marcus, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," IEEE Trans. Softw. Eng., vol.33, no.6, pp.420–432, 2007. [doi: 10.1109/TSE.2007.1016]

[12] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," J. Systems and Software, vol.79, pp.57–68, 2006. [doi: 10.1016/j.jss.2004.12.018]

[13] O. Panchenko, "Concept location and program comprehension in service-oriented software," Proc. 23rd International Conference on Software Maintenance, pp.513–514, 2007. [doi: 10.1109/ICSM.2007.4362676]

[14] J.V. Geet and S. Demeyer, "Feature location in COBOL mainframe systems: An experience report," Proc. 25th International Conference on Software Maintenance, pp.361–370, 2009. [doi: 10.1109/ICSM.2009.5306312]

[15] A. Rohatgi, A.H. Lhadj, and J. Rilling, "Approach for solving the feature location problem by measuring the component modification impact," IET Software, vol.3, no.4, pp.292–311, 2009. [doi: 10.1049/iet-sen.2008.0078]

[16] K. Chen and V. Rajlich, "Case study of feature location using dependency graph, after 10 years," Proc. 18th IEEE International Conference on Program Comprehension, pp.1–3, 2010. [doi: 10.1109/ICPC.2010.40]

[17] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," Proc. 15th IEEE International Conference on Program Comprehension, pp.37–48, 2007. [doi: 10.1109/ICPC.2007.13]

[18] IBM Corporation , http://www.alphaworks.ibm.com/tech/biptk?open &S_TACT=105AGX59&S_CMP=GR&ca=dgr-lnxd02awbiptk, access at Oct. 2010.

[19] Jhotdraw, http://sourceforge.net/projects/jhotdraw/, access at Oct. 2010.

[20] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," Proc. 18th International Conference on Program Comprehension, pp.14–23, 2010. [doi 10.1109/ICPC.2010.10]

[21] D.P. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering, 2007. [doi: 10.1145/1321631.1321667]

[22] M. Revelle, "Supporting feature-level software maintenance," Proc. 16th IEEE Working Conference on Reverse Engineering, pp.287–290, 2009. [doi: DOI 10.1109/WCRE.2009.43]

[23] R. Sindhgatta, "Using an information retrieval system to retrieve source code samples," Proc. 28th ACM International Conference on Software Engineering, pp.905–908, 2006. [10.1145/1134285.1134448]

Zhengong Cai received the B.S. degree in computer science from Zhejiang University in 2006. He is currently a PHD student in the College of Computer Science, Zhejiang University. His research interests include software reengineering, domain modeling and component-based software engineering.

Xiaohu Yang received a PhD degree in computer science from Zhejiang University in 1993. Since 1994, he has been a faculty member and associate professor in the College of Computer Science, Zhejiang University. His research interests include software engineering, legacy system reengineering, and software technology financial services. He is a member of the IEEE and the IEEE Computer Society.

Xinyu Wang (BSc, PhD) is currently a faculty member and lecturer in the College of Computer Science, Zhejiang University. His research interests include software engineering, distributed software architecture and distributed computing. He is a member of IEEE Computer Society.

Aleksander J. Kavs received his BSc degree in occupational safety from the University of Nis, Yugoslavia in 1971. After doing operational work in the field and teaching at the School of Safety Engineering at the University of Ljubljana, Yugoslavia, he moved to the US in 1981 and since then he works in the software development. His research interests include designing and programming database management systems, computer languages and workflow and messaging systems to financial applications and document imaging. Currently he heads several development groups at State Street Hangzhou.