Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval

Denys Poshyvanyk, *Member, IEEE*, Yann-Gaël Guéhéneuc, *Member, IEEE*, Andrian Marcus, *Member, IEEE,* Giuliano Antoniol, *Member, IEEE,* Václav Rajlich, *Member, IEEE Computer Society*

Abstract—This paper recasts the problem of feature location in source code as a decision-making problem in the presence of uncertainty. The solution to the problem is formulated as a combination of the opinions of different experts. The experts in this work are two existing techniques for feature location: a scenario-based probabilistic ranking of events and an information retrieval-based technique that uses latent semantic indexing. The combination of these two experts is empirically evaluated through several case studies, which use the source code of the Mozilla Web browser and the Eclipse integrated development environment. The results show that the combination of experts significantly improves the effectiveness of feature location when compared to each of the experts used independently.

Index Terms—program understanding, feature identification, concept location, dynamic and static analyses, information retrieval, Latent Semantic Indexing, scenario-based probabilistic ranking, open source software.

1 Introduction

SOFTWARE evolution requires adding new functionalities to programs, improving existing functionalities, and removing bugs, which can be considered as a removal of unwanted functionalities. A feature represents in a program some functionality that is accessible by and visible to the developers and usually it is captured by explicit requirements. Identifying the parts of the source code that correspond to a specific functionality is a prerequisite to evolution and is one of the most common activities undertaken by developers. This process is called *feature* (or *concept*) *location* [1] and it is a part of the incremental change process [2].

Although incremental change ultimately needs to identify all components to-be-changed, the programmer must find the location in the code where the first change must be made. For that, the programmer uses a search process where the search space is the whole program and where various search techniques narrow down the search space. The literature limits this step to finding small number of feature components and it is considered sufficient for a successful feature location to find only one such component. The full extent of the change is then handled by impact analysis, which starts where concept location ends and finds all remaining impacted components. Methodologically the two activities are different from each other and this is the reason why they are treated separately in the literature [3], [4], [5], [6], [7], [8]. Different types of techniques are effective in one activity versus the other. In this work, we are specifically addressing the identification of methods in object-oriented software that are part of the implementation of a feature (i.e., they change when the feature is altered) and can be used as starting points in impact analysis.

While developers often perform feature location manually, tool support is needed for large and complex programs. Existing tools supporting feature location rely on data collected by static and–or dynamic analysis of the program. Dynamic feature location is based on collecting and analyzing execution traces, which identify methods that are executed for a specific scenario. Unfortunately, dynamic analyses are often unable to distinguish between overlapping features, because the same methods may contribute to several features. Static analyses better filter and organize data but they can rarely identify methods contributing to a specific execution scenario exactly. The research community has long recognized the need to combine static and dynamic techniques into hybrid techniques to improve the effectiveness of feature location [9], [10], [11], [12]. The general benefits of combining static and dynamic analyses, for example described by Ernst [13], are that each kind of analysis collects data that

- *D. Poshyvanyk, A. Marcus and V. Rajlich are with the Department of Compute Science at Wayne State University, Detroit, MI, 48202 E-mail: (denys, amarcus, rajlich)@cs.wayne.edu.*
- *Y-G. Guéhéneuc is with the Department of Experimental Software Engineering, University of Montreal, Canada. E-mail: guehene@iro.umontreal.ca*
- *G. Antoniol is with the Département de Génie Informatique, École Polytechnique de Montréal, Canada. E-mail: antoniol@ieee.org*

Manuscript received 11/08/2006.

would be unavailable to the other analysis but could improve its performance. Indeed, static analyses are conservative and sound while dynamic analyses are efficient and precise (given appropriate test suites). Thus, combining static and dynamic analyses compensate the imprecision of static analyses and the unsoundness of dynamic analyses.

All current hybrid techniques share a common assumption: the fact that a method belongs to an execution trace or that a module is activated by a feature can be determined with a complete certainty and thus can be expressed with a Boolean value. However, in reality, imprecision in the measures because of scheduling and sampling, uncertainty in the environment, perturbations in the execution due to multi-tasking, or simply lack of knowledge, lead towards expressing the relation between scenarios and features in nondeterministic terms [14]. When a specified scenario is executed, a deterministic relation between the trace and the scenario exists, yet we cannot be certain whether a method call and–or a field access is relevant to the feature. Thus, we have a nondeterministic relation between scenarios and features.

We use both certain and uncertain data extracted respectively with static and dynamic analyses and filtered by probabilistic and information retrieval techniques to identify features in source code. We use two previously developed techniques for feature location that provide complementary results. The first technique is based on Latent Semantic Indexing (LSI) [15] of the source code [16]. The second technique is a Scenario-based Probabilistic Ranking (SPR) of events observed while executing the program under given scenarios [12], [17]. We choose these techniques because they have shown promising results and are actively developed in-house.

Using LSI, developers can query static documents (i.e., code and documentation) and obtain a ranked list of code fragments relevant to a feature. Using SPR, developers analyze dynamic traces resulting from the execution of different scenarios and obtain a list of entities (i.e., methods and classes), again ranked according to their relevance to a feature of interest.

By definitions, both the LSI-based and the SPR-based techniques provide an uncertain judgment. We draw a parallel between them and two collaborating *experts*, who provide their judgments independently. We are in the position of a manager who must combine these two experts' subjective forecasts [18], [19] to increase the certainty of the result. We use an affine transformation [18] to combine the two experts' judgements. The affine coefficient expresses our confidence in each expert and its ability to identify features correctly. The proposed combination of experts provides a new feature location technique named PROMESIR (**P**robabilistic **R**anking **O**f **M**ethods based on **E**xecution **S**cenarios and **I**nformation **R**etrieval).

We show that feature location with PROMESIR outperforms the judgment of either single expert. We perform the comparison through several case studies. First, we apply PROMESIR to the scenarios presented in an earlier case study [12] and the results clearly show the superiority of the new method. Then, we identify methods and classes involved in several documented bugs. We compare the methods and classes identified by PROMESIR with those actually contained in the official patches for fixing the bugs. We show that PROMESIR identifies the relevant methods with better accuracy than either of the two techniques individually.

The remaining sections of this paper are organized as follows. Section 2 presents an overview of related work on dynamic and static techniques for feature location. It also briefly introduces background information on the LSI-based and SPR-based techniques. Section 3 presents our new technique for feature location, namely PROMESIR. The evaluation of the new technique with several case studies is presented in Section 4. The conclusions and future work are outlined in Section 5.

2 Previous Work

Existing techniques for feature location broadly fall into three categories, based on the type of information they use: dynamic, static, and hybrid.

Software reconnaissance by Wilde et al. [20] was the first published dynamic technique to identify features by analyzing execution traces of test cases. Two sets of test cases are used to build two execution traces: an execution trace where the desired feature is exercised and an execution trace where the feature is not exercised. The two traces are compared to identify the entities of the program that implement the feature. This technique was recently extended to improve its accuracy [12], [14], [21] by introducing new criteria on selecting execution scenarios and by analyzing the execution traces differently. Similarly, Wong et al. [22] analyzed execution slices of test cases to identify features in the source code.

Biggerstaff et al. [23] introduced static feature location as the "concept assignment problem" and designed a tool that utilizes parsing, simple clustering, identifier names, and a browser, to support the identification. The simplest and most used static techniques are based on searching the source code using text pattern matching tools, such as Unix *grep* [24]. A significant improvement over the *grep*-based tools are the information retrieval-based approaches [16], which provide ranked results to the developer's queries. Chen and Rajlich [25] proposed a technique for feature location based on searching the Abstract System Dependence Graph (ASDG). This process is improved in [26], where the search of the dependency graph is guided based on the analysis of the topology of the structural dependencies. Some methods combine different kinds of static information (i.e., lexical and structural), such as the one proposed by Zhao et al. [27], which uses information retrieval and a branch-reserving call graph to search the source code. A comparison of static feature location techniques is presented in [28].

Eisenbarth et al. [9] combined both static (i.e., dependencies) and dynamic (i.e., execution traces) data to identify features in programs. They use formal concept analysis to relate features together. Salah and Mancordis [29] also use static and dynamic data to identify features in

Java programs.

In the remainder of this section, we further introduce the LSI-based (Section 2.1) and the SPR-based (Section 2.2) techniques, which form the foundation of our new PROMESIR technique.

2.1. Information Retrieval-based Feature Location using Latent Semantic Indexing

Static feature location is essentially a search task, whether the developer searches the source code or the documentation and most information retrieval techniques, including LSI, facilitate this search. LSI is an advanced information retrieval method that analyzes the relation between words and documents in large bodies of text. For every document in the text, the technique generates a real-valued vector description. This representation can be used to index and to compare documents, using different similarity measures. The central concept of LSI is that information about contexts in which a particular word appears or does not appear provides a set of constraints that determines the similarity of the meanings of words with one another. By applying LSI to source code and its internal documentation (i.e., comments), candidate components can be compared using these similarity measures. More technical details on LSI are available in [15]. We chose LSI over vector space models (VSM) or other IR methods for two main reasons. First, LSI has been shown to address the problems of polysemy and synonymy [15] quite well, which is important with respect to the feature location problem, because developers usually construct queries without knowing precisely the vocabulary of the target system. Second, in our previous work on traceability link recovery [30], we showed that LSI outperforms VSM and Bayes classifiers.

Internal source code documentation has been recognized as commonly written in a subset of English [31] that lends itself to the use of information retrieval techniques. In addition, identifiers in the source code form a language of their own with no defined grammar or morphological rules. LSI is well-suited to deal with such situations, as it does not use a predefined grammar or vocabulary. The meanings of the words in LSI are derived from their usage rather than from a dictionary or thesaurus, which is an advantage over a traditional natural language approach [31], where a subset of the English grammar and a dictionary must be developed.

In software engineering, LSI has been used for many tasks closely related to feature location, such as reuse [32], [33], [34], abstract data types identification [35], clone detection [36], traceability link recovery between source code and external documentation [30], [37], conceptual coupling [38] and cohesion [39], clustering [40], requirements tracing [41], establishing traceability links between artefacts [42], etc.

Marcus et al. [16] proposed an information retrieval-based technique for feature location using LSI. Using LSI for feature location allows the developer to formulate queries in natural language (or a close format) and the results are returned as a list of source code elements, ranked by the relevance to the query. As an analogy, LSI is used for feature location in source code much like today's popular search engines (e.g., Google) are used to search the Web or local hard drives.

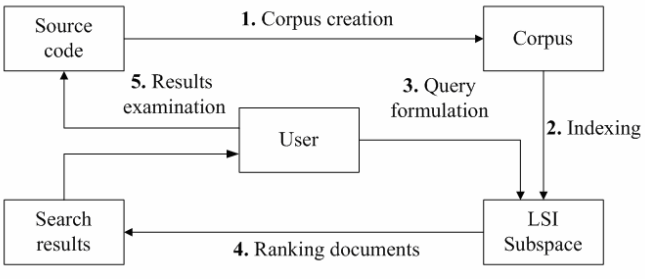

Fig. 1. The feature location process using LSI

Figure 1 shows the main steps in the feature location process using LSI. Tool support exists to use this method in MS Visual Studio for C++ projects [43] and in Eclipse for Java projects [44]. The process has five major steps:

1. Corpus creation. The source code is parsed using a developer-defined granularity level (i.e., methods or classes) and documents are extracted from the source code. A corpus is created to be indexed with LSI, such that each method (and–or class) will have a corresponding document in the corpus.

2. Indexing. The corpus is indexed using LSI and a representation of the corpus as a real-valued vector subspace is created. In this subspace, each document (hence each method or class) has a corresponding vector.

3. Query formulation. A developer selects a set of words that describe the feature. This set of words constitutes the query. The tool checks whether the words from the query are present in the vocabulary of the source code (generated by LSI). If no word is present, then the tool:

a. Suggests similar words using the vocabulary of the source code.

b. Eliminates the word from the initial query. If the elimination of a word significantly alters the meaning of the query, step 3 is reapplied with additional words for the query.

4. Ranking documents. Similarities between the query and every document from the source code are computed. We use the cosine between the vectors corresponding to the query and a document as similarity measure. This similarity measure yields values between [-1, 1] for any pair of vectors, with 1 corresponding to identical documents. Negative values are associated with unrelated documents. The similarity between a query expressing some semantic characteristics of a feature and a set of data about the source code (e.g., manual pages, documentation, classes, or methods) indexed via LSI enables the production of a ranking of documents relevant to the feature. All the documents are ranked by the similarity measure in descending order. The final score for each document x and the current query is denoted as $f_{lsi}(x)$.

5. Results examination. The developer examines the source code documents sorted by $f_{lsi}(x)$, starting with documents with highest similarities. For every source code document examined, a decision is required whether the document is part of the feature or not. If it is part of

the feature, then the search succeeded, if it is not and new knowledge obtained from the investigated documents helps formulate a better query (e.g., narrow down the search criteria), then step 3 should be reapplied else the next document in the list should be examined.

In our experience, developers tend to look at only the few first documents before they formulate new queries [16], [28], [45], [46]. Detailed examples on how this method is used can be found in Section 4.5 and in [16], [28].

2.2. Feature Location using Scenario-based Probabilistic Ranking

We present for the sake of completeness the SPR-based technique, previously published in [12], [17]. SPR is a dynamic analysis technique that links features with program source code and thus identifies entities of the program implementing the specific feature using different scenarios.

A **scenario** defines the context in which a feature is studied, for example the sequence of the developer's actions with the program. Let F' (and F respectively) be a set of scenarios (not) exercising a feature of interest; scenarios are used to collect traces. A **trace** is a sequence of **events**, dynamically collected by executing a given scenario. Events are traditionally defined loosely [17] but, in the context of this work, they correspond to method and function calls. An **interval** is a subsequence of contiguous events belonging to a **trace**. Intervals may or may not pertain to a feature and thus may or may not contain relevant events. A **feature of interest** is characterized dynamically by the events related to it and statically by its **micro-architecture**, which is a collection of entities activated by the feature (i.e., classes, interfaces, fields, and methods).

We want to classify events as relevant or irrelevant with respect to the many *possible* different features intentionally or *accidentally* exercised in the sets of scenarios F' and F. Exercising scenarios in F' produces the class $C_{I'}$, a set of intervals I'_i containing events relevant to the feature. Scenarios in F produce C_I, a class of intervals I_j containing events irrelevant to the feature. Intervals I'_i (I_j, respectively) are marked as relevant (irrelevant, respectively) during trace collection. However, they may contain irrelevant (relevant, respectively) events due to uncertainty. Yet, relevant events have frequencies higher in $C_{I'}$ than in C_I intervals. Consequently, the problem of deciding if an event e_k contributes to a feature is mapped to the problem of testing if the event e_k is statistically more frequent in the I'_i intervals than in the I_i intervals.
For any given interval I (either I'_i or I_j), the e_k frequency is computed as the ratio $N_I(e_k) / N_I$, where $N_I(e_k)$ is the number of times e_k appears in I and N_I is the overall number of events in I. We classify events as relevant when their frequencies in intervals I'_i is higher than in intervals I_j. The classification process is parameterized by a threshold Θ and a level of confidence a. For each pair of intervals in $C_{I'}$ and C_I and for each event we perform hypothesis testing attempting to reject the null hypothesis H_0 with a confidence level of a, that the event frequency is the same in two intervals I'_i and I_j. If the null hypothesis is rejected more than Θ times the event is tagged as relevant to the feature. We then use a relevance index to rank and associate the most relevant events with the feature of interest. Let e_i be an event classified as relevant and let $N_{C_{I'}}(e_i)$ be the number of times e_i appears in all intervals belonging to $C_{I'}$ ($N_{C,(e_i)}$ the number of times it appears in $C_{I'}$) further assume that $N_{C_{I'}}$ is the overall number of events in $C_{I'}$ (N_{C_I} the overall number of C_I events), then the score referred to as $f_{spr}(e_i)$ is computed as:

$$f_{spr}(e_i) = \frac{N_{C_{I'}}(e_i) \Big/ N_{C_{I'}}}{N_{C_{I'}}(e_i) \Big/ N_{C_{I'}} + N_{C_I}(e_i) \Big/ N_{C_I}} \quad (1)$$

Several events may have the same score and the number of relevant events might be too large to be of any help to developers, therefore we filter ranked relevant events and keep only those with a higher relevance. We use Equation 2 and a positive threshold t to reduce the set of feature-relevant events and thus to limit the number of events that developers must consider as relevant to a feature of interest:

$$E'_t = \{e_i \mid f_{spr}(e_i) \geq t\} \quad (2)$$

We extend the definition of f_{spr} in Equations 1 and 2 as applying to the methods and–or functions corresponding to events rather than to the events themselves. We thus can directly compute the score of a method or a function.

Then, we use the subset E'_t of ranked relevant methods and functions to build a micro-architecture representing the feature of interest, which can be displayed against the program architecture and compared to other micro-architectures to help developers precisely locate responsibilities.

Results reported in the following case studies were obtained by selecting an acceptance threshold value of $N_{C_{I'}} \times N_{C_I} - 1$, and a of 1.00%; these values, in our experience, ensure relatively small sets E'_t while retaining most relevant events.

3 PROMESIR: COMBINING THE EXPERTS

We propose a new technique to improve the precision of feature location by combining the LSI- and SPR-based techniques into PROMESIR. We consider the LSI and SPR rankings of source-code elements as the judgments of two independent experts, who provide their expertise to solve the problem of identifying a feature. We draw inspiration from Jacobs [18] to combine the LSI and SPR rankings via an affine transformation, using the following equation:

$$f(x) = \sum_{i=1}^{n} \lambda_i \beta_i f_i(x) \quad (3)$$

where $f_i(x)$ is the judgment of the i^{th} expert, λ_i is a weight expressing our confidence in the i^{th} expert and β_i is a re-normalization constant. We use re-normalization con-

stants because the different experts may express judgments that are not commensurable. The λ_i coefficient can also be selected so that $f_i(x)$ lies in the interval [0, 1], as we do in our case. The judgments of the experts should be in the same interval, thus imposing the constraints that the weight λ_i defines an affine transformation: $\sum_{i=1}^{n} \lambda_i = 1$

Our experts state their respective judgments based on different data. The LSI expert builds its judgment based on the source code lexical similarities, while the SPR expert grounds its judgment on the probabilistic ranking of dynamic events observed in execution traces. Yet, both experts answer, through different means, the same question: "*What is the location of a feature of interest?*" PROMESIR combines the valuable expertise of both experts to obtain a more accurate answer and to minimize the developer's effort.

We must align the definitions used in each technique to formally define the PROMESIR technique. Let x be an entity (method or class in an object-oriented program or a function when analyzing non object-oriented programs) declared in the implementation of a program. As previously mentioned, we denote with $f_{lsi}(x)$ and $f_{spr}(x)$ the score assigned to x by our experts, LSI and SPR respectively. The $f_{lsi}(x)$ and $f_{spr}(x)$ scores are not defined over a same domain: $f_{lsi}(x)$ score takes values in [-1, 1] while $f_{spr}(x)$ score is defined over [0, 1]. The combination of the two experts' judgments with Equation 3 requires a re-normalization of their scores. This normalization must not disrupt the LSI and SPR experts' judgments because we want to promote entities that both experts consider relevant. Among several possible renormalizations, we select a very simple transformation grounded on the fact that the negative values of the LSI-based similarities are irrelevant. The re-normalized LSI and SPR scores are obtained as follows:

$$\beta_{spr}(x) = 1$$

$$\beta_{lsi}(x) = \begin{cases} 1 & if\ f_{lsi}(x) > 0 \\ else & 0 \end{cases} \quad (4)$$

The combination of the two experts in PROMESIR leads to rewrite Equation 3 as:

$$f_{PROMESIR}(x) = \lambda \beta_{spr} f_{spr}(x) + (1-\lambda)\beta_{lsi} f_{lsi}(x) \quad \textbf{(5)}$$

where λ expresses our confidence on the respective ability of the LSI or SPR experts to score entities relevant to a feature of interest with better accuracy. We use the scores $f_{PROMESIR}(x)$ to sort source code entities and present the ranked list to the developers, who examine the source code entities starting with those with highest scores and in descending order, until entities most relevant to the feature are found.

Given this combination method, the results of one expert do not depend on the other. Thus, PROMESIR allows for LSI and SPR to be run in parallel, if so desired. Both experts need user input, but they operate differently, as presented above: LSI is based on an iterative approach, where results are improved at each step, whereas SPR is non-iterative. Consequently, PROMESIR is also an iterative technique as follows:

1. Compute $f_{spr}(x)$ as described in Section 2.2.
2. Run steps 1, 2, 3, 4 from the LSI-based approach as described in Section 2.1.
3. Combine the rankings based on Equation 5.
4. Examine the results as in step 5 of the LSI-based method.

In the following case studies, we applied each technique in parallel, so we can assess their accuracy individually. In this case PROMESIR is not applied iteratively, but the combination of judgments is done on the best result of each expert.

4. Evaluation of PROMESIR

We performed several case studies to assess the effectiveness of PROMESIR. All the case studies are designed and conducted according to recommendations in the state-of-the-art [47]. We present a case study replicating a previously published experiment [12], further developed in [17]. We show that PROMESIR, on average, is more effective than either LSI or SPR.

In each task, we compare the rankings produced by LSI, SPR, and PROMESIR. We conclude with a discussion on the increase of effectiveness and on the choice of the queries and scenarios.

4.1. Design and Objectives of the Case Studies

In the following case studies, we chose methods as the level of granularity for the software entities and we assess and compare the effectiveness of the three techniques when identifying a *first* method relevant to a feature of interest. We focus on the first identified method because the other methods implementing the feature are inferred during the design of a change, which includes impact analysis and change propagation. Thus, we set the goal of each case study as the accurate location of at least one method that implements part of the feature.

We also chose large open-source programs to show the scalability of our techniques and to allow replication of the studies. In each case study but the first one, we use documented bugs to assess our work: each documented bug is used as a *gold standard* against which we compare the results of the techniques. Indeed, the documentation of each bug specifies which methods were changed to fix this bug. We consider these changed methods as belonging to the *unwanted feature* associated with the bug. One method may belong to more than one bug (i.e., changed in different bug fixes), but it is at least exercised in the associated unwanted feature. We do not attempt to identify defects (i.e., the root cause of a bug) such as a loop off by one, because we work at the method level and have therefore no information on the executed statements. We used the following criteria to select bugs for the case study:

- We chose well-known, documented, and reproducible bugs;
- We selected bugs that do not include methods and classes involved in the first case study to prevent obtaining better results by chance;
- We chose bugs with available and approved patches applied in recent releases.

None of the authors knew the parts of the programs corresponding to the features to eliminate potential bias. In particular, the evaluations of the relevance of each technique is founded solely on the methods identified by the techniques and on the methods present in the official patches associated with each bug, thus as experimenters, we did not have to make any subjective decisions biasing the results of our evaluations.

In the following case studies, we use a λ value of 0.5. We investigate the influence of the λ value on the PROMESIR ranking in the discussions.

4.2. Objects of the Case Studies

We use two large open-source programs for our case studies: Eclipse[1] and Mozilla[2]. Eclipse is an open-source integrated development environment. It is a platform used both in the open-source community and in industry, for example as a base for the WebSphere family of development environments. Eclipse is mostly written in Java, with C/C++ code used mainly for the widget toolkit, which we do not analyze. We use versions 2.0.0, 2.1.0, 2.1.3, 3.0.1, 3.1.1, and 3.3M1. For example, version 2.1.3 contains 7,648 classes with 89,341 methods in more than 8,000 source code files for about 2.4 MLOC (millions lines of code).

Mozilla is an open-source Web browser ported to almost all known software and hardware platforms. It is as large as many industrial size programs and it is developed mostly in C++. We do not analyze the parts of Mozilla written in other programming languages, like C, Java, IDL, XML, HTML, etc. In our case studies, we use the source code of versions 1.5.1 and 1.6 of Mozilla. Version 1.5.1 includes 4,853 classes and 53,617 methods implemented in more than 10,000 source files and 3,500 different subdirectories, for about 3.7 MLOC.

We choose to use conservative reverse-engineering techniques and apply strict rules classifying as classes only entities declared as such according to the C++ syntax. Moreover, we consider structures and complex templates (e.g., templates mixed with structures) as outside of the boundary of reverse-engineered models and do not recover their attributes, methods, and locations in source code files. Different tools were used to extract information from Mozilla and Eclipse [48].

In each study, we follow the method for LSI ranking described in Section 2.1. We construct the corpus for each software system by extracting all comments and identifiers from the source code. The resulting text is processed as follows: some tokens are eliminated (e.g., operators, special symbols, some numbers, keywords of the programming language, standard library function names, etc.); the identifier names in the source code are split into parts based on known coding standards whereas the original form of each identifier is preserved as well; a document of the corpus is created with the comments and identifiers corresponding to one method. We do not use a predefined vocabulary or a predefined grammar, therefore no morphological analysis or transformations are applied. Some researchers use word stemming, however this is an optional step, which is not required while using LSI. Table 1 shows the size of the corpora we created. Based on our previous experience [49], we use a dimensionality reduction factor of 500, which accurately represents a semantic space of this size.

TABLE 1
LSI CORPUS VITALS FOR MOZILLA AND ECLIPSE

	Mozilla 1.5.1	Eclipse 2.1.3
MLOC	4.4	2.9
Vocabulary	85,439	56,861
Number of parsed documents	68,190	86,206

Similarly, the use of the SPR technique follows the pattern described in Section 2.2. The SPR technique provides sets of ranked methods using their frequencies in the execution traces.

4.3. Measuring the Effectiveness of the Techniques

We need a uniform measure to compare the effectiveness of the feature location techniques. This measure must be suited to evaluate the results of all LSI, SPR, and PROMESIR techniques. We do not use standard information retrieval measures such as precision and recall, because the feature location techniques may assign scores to *all* methods in a program, consequently, and without a threshold, recall always equals 1.0 while precision equals *1/k*, with *k* the number of methods in the program. We could use a threshold to include in the precision and recall only methods with a score higher than the threshold. However, this would artificially increase the number of parameters of our techniques.

Thus, to compare the techniques, we use the rank of the first related method as a measure of effectiveness of every technique. Every technique, LSI, SPR, and PROMESIR outputs a ranked list of methods that developers must investigate. The goal of each technique is to reduce the effort of the developers in the location process. Thus, we measure the effort that the developers must spend as the number of methods from the list that they need to investigate until they find the first related method. We do not take into account the size of the software system, because in every case the developer must investigate *n* methods before finding the first relevant method. In analogy, consider an Internet search engine, which provides the list of results in response to a user's query. The searching effort may be measured as the number of links she must visit before she finds relevant information. Note that the size of the World Wide Web grows every day while this fact successfully remains latent from the user's perspective.

[1] www.eclipse.org
[2] www.mozilla.org

TABLE 2
METHODS RESPONSIBLE FOR THE UNWANTED FEATURES IN ECLIPSE

Bug #	Methods
5138	org.eclipse.jdt.internal.ui.text.java.JavaStringDoubleClickSelector.doubleClicked org.eclipse.jdt.internal.ui.text.java.JavadocDoubleClickStrategy.doubleClicked org.eclipse.jface.text.TextViewer.mouseDoubleClick org.eclipse.jface.text.TextViewer.mouseUp org.eclipse.jface.text.DefaultTextDoubleClickStrategy.doubleClicked org.eclipse.jface.text.DefaultTextDoubleClickStrategy.wordPosition
31779	org.eclipse.core.internal.localstore.UnifiedTree.addChildren org.eclipse.core.internal.localstore.UnifiedTree.addChildrenFromFileSystem org.eclipse.core.internal.localstore.UnifiedTree.createChildNodeFromFileSystem
74149	org.eclipse.help.internal.search.QueryBuilder.tokenizeUserQuery

Formally, we define the effectiveness of a technique j, E_j, as the rank $r(m_i)$ of the method m_i, where m_i is the top-ranked method among the methods that must be changed to fix a bug (i.e., implementing part of the located feature). The effectiveness measure shows how many methods must be investigated before the first method relevant to the feature is located. Obviously, in the best case scenario, the developer will find one of the desired methods in the first place, so she only needs to investigate that method, hence the efficiency value is one (1). A higher effectiveness value indicates more search effort needed.

While the effectiveness measure can be applied to the results of LSI and PROMESIR without any problems, we need to make certain assumptions for SPR. The SPR technique may score many methods of the source code as 100% relevant to a feature of interest (i.e., $r(m_i) = 1$), while not all of them are in fact relevant. Thus, to define the SPR effectiveness measure, we follow the *average case scenario* in which developers must investigate half of the methods from the set of methods ranked as 100% relevant to the feature.

Regardless of the technique used to identify features, eventually the developer must inspect a number of methods until she finds one method that actually implements part of the feature. Effectiveness measures how many methods the developer must inspect to identify a starting point of the feature. The goal of each technique is to assist the developer such that she needs to look at fewer methods. The effectiveness can be regarded as an inverse measure as the lower the rank of the first method the better the result is. In other words, low effectiveness value shows that less search effort is needed to locate the starting point of feature implementation.

4.4. The First Case Study – Book-marking in Mozilla

This first case study replicates a previous case study [12], where the feature to be located is "*save a bookmark*", which can be stated as: "*identifying methods in Mozilla that are part of the feature activated when a URL is saved*". We perform the same case study (i.e., the same scenario and the same feature location task) to compare the previous results from SPR ranking alone with results from LSI and PROMESIR. This replication is important to compare the three ranking techniques with one another. It is a *partial* replication because we replicate the scenario and the task only – we do not apply again the SPR ranking, as the results are available elsewhere [17].

We considered two scenarios in which a developer is interested in understanding the inner workings of Mozilla regarding its book-marking feature:

- *Scenario 1*: The developer visits a URL. She opens Mozilla, clicks on a previously bookmarked URL, waits for the page to load, and closes the browser;
- *Scenario 2*: The developer acts as before but, once the page is loaded, she saves the URL using the mouse right button and closes the Web browser.

We applied the SPR ranking by running Mozilla according to the two scenarios and by collecting corresponding dynamic traces [17]. We then produced sets of relevant methods to the feature of interest. LSI rankings were computed by formulating a query with the terms related to "*bookmark*" from the vocabulary of Mozilla generated during indexing with LSI. We used our judgment to assess whether the terms relate to the feature of creating a new bookmark. We created the following query: "*bookmark newbookmark bookmarkname bookmarkresource bookmarkadddate createbookmark insertbookmarkitem deletebookmark bookmarknode*". Three methods (identified via manual inspection) are relevant to this feature, all from the class *nsBookmarkService*: *AddBookmarkImmediately*, *CreateBookmark*, and *CreateBookmarkInContainer*. LSI ranked these methods on the 1st, 14th, and 36th positions respectively. No ranking is reported for SPR because the methods responsible for implementing the feature were scored 100% relevant among 272 methods and thus could be assigned any arbitrary rank between 1 and 272. PROMESIR ranks these three methods on the 1st, 2nd, and 4th position respectively. The ranks of the three methods highlights that LSI and SPR are complementary techniques. Indeed, none of the other 33 methods ranked by LSI do not appear in the 272 methods from SPR.

The effectiveness for LSI, SPR, and PROMESIR respectively, are: $E_{LSI} = 1$, $E_{SPR} = 272 / 2 = 136$, and $E_{PROMESIR} = 1$.

The values of effectiveness for LSI and PROMESIR are equal because we only consider the rank of the first top method. These results provide evidence that the combination of experts is likely to improve individual rankings.

TABLE 3
SCENARIOS AND QUERIES USED FOR ECLIPSE

Bug #	Scenario 1 (Not exercising the bug)	Scenario 2 (Exercising the bug)	LSI Query
5138	User starts Eclipse and edits a piece of text in the default editor without performing any drag-and-drop.	User starts Eclipse, edits a piece of text and performs the steps to reproduce the bug: Click once and release the mouse button Click a second time quickly, and hold the mouse button down. Drag and select some text. Release the mouse button. The text you have selected is deselected.	*mouse double click up down drag release select text offset document position*
31779	User starts Eclipse and creates resources, such as text file and folder and performs refresh operations on projects.	User starts Eclipse and creates resources. She also creates a file using the file system in a project and then performs a refresh operation.	*unified tree node file system folder location*
74149	User starts Eclipse, brings the help forward, and perform a simple query with no quotes or properly quoted text, such as "Web browser"	User starts Eclipse, brings the help forward, and perform a query, leaving the last quoted text unclosed, such as "Web browser	*search query quoted token*

The following case studies are much more extensive, ranging across application domains and programming languages, and have the goal of verifying the findings of the first case study.

4.5. The Second Case Study – Locating Bugs in Eclipse

We applied the three techniques to locate three different unwanted features (i.e., bugs) in Eclipse. Each bug is chosen for its characteristics that satisfy the previously described criteria. These bugs are:

1. Bug #74149[3], described as *"The search words after ' " ' will be ignored"*, present in the versions 3.0.0, 3.0.1, 3.0.2, and fixed in version 3.1.1.
2. Bug #5138[4], described as *"Double-click-drag to select multiple words doesn't work"*, present in version 2.1.3 and fixed in version 3.3M1.
3. Bug #31779[5], described as *"UnifiedTree should ensure file/folder exists"*. This bug exists in the version 2.0.0 and subsequently fixed in 2.1.0.

Methods and functions modified by developers to remove the bugs were identified by inspecting the provided patches recognized by the Eclipse source code reviewers. Table 2 reports the patched methods that have been actually modified to fix those bugs. Very often more than one patch was produced in the process of fixing a bug. In such a case, we considered both the first patch, often corresponding to a quick answer to the urgent need to fix the bug, and the last patch, usually involving careful reorganization or even refactoring of several classes and methods. For example, for the bug #5138 there are two patches, thus we considered the union of the modified methods from both patches. We did not consider intermediate patches, as they are always superseded by the last patch and may contain spurious changes.

In Table 3 we list descriptions of two families of scenarios (one exercising the feature and one not) used to obtain the SPR rankings and the queries which were executed to obtain the best LSI rankings of the methods for each Eclipse bug.

In each instance the scenarios are extracted from the bug description. Selecting the initial queries and refining them is based on the experience of the developer. A naïve approach is to simply use the entire bug description as initial query and then refine it, based on the results or select several related terms from the bug description. We choose to start from simple queries consisting only of a few terms and this approach resulted in formulating at least one extra query each time and at most three extra queries. Table 3 shows only the last performed query, which generated the best LSI ranking. We show in detail how the queries were formulated for bug #74149 to better understand this approach.

Based on the bug description and the developer's interpretation, using the method explained in Section 2.1, the initial query is formulated as *"search query"*. The results did not return any relevant method in the top of the ranked list.

Given these results, we reformulated the query to be more specific to our search. We noticed that the word "token" is used in methods that deal with parsing a query and the Eclipse vocabulary indicates the existence of the word "quoted". We combined these words into a new query, "search query quoted token", which returned one relevant method in the 5th position (see Figure 2 and Table

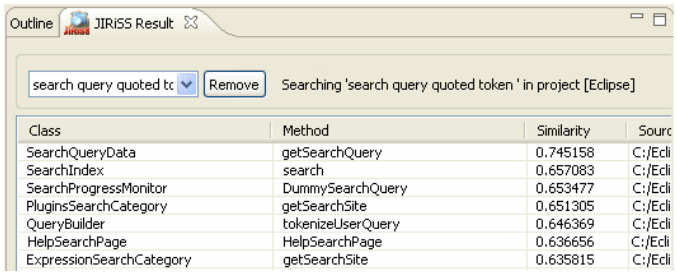

Fig. 2. The list of ranked results for the query 'search query quoted token' for bug#74149

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=74149
[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=5138
[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=31779

TABLE 4
EFFECTIVENESS OF EACH TECHNIQUE FOR THE ECLIPSE BUGS

Bug #	E_{LSI}	$E_{PROMESIR}$	E_{SPR}	Method name	PROMESIR over LSI	PROMESIR over SPR
5138	7	1	268	JavaStringDoubleClickSelector.doubleClicked	7×	259×
31779	2	1	170	UnifiedTree.createChildNodeFromFileSystem	2×	165×
74149	5	3	456	QueryBuilder.tokenizeUserQuery	2×	156×

3). For the first query "search query" the top four returned methods are the same as for the second query. We used similar judgments in our case studies. For simplicity, we only report the final query from here on.

Table 4 reports the name of the method first identified as relevant to the bug and its ranking by each method. As explained before, the SPR ranking is in fact the number of methods that are considered 100% relevant to the feature of interest by this technique, divided by two, because we consider average case scenario. We also compute the improvement that PROMESIR brings over the individual technique. Based on the results in Table 4 we conclude that combining the two LSI and SPR techniques dramatically improves the effectiveness of the feature location.

4.6. Third Case Study – Locating Bugs in Mozilla

We applied the three techniques to locate five different bugs in Mozilla. Each bug is chosen for its characteristics that satisfy the previously described criteria. These bugs are:

1. Bug #182192[6], described as "*quotes (") are not removed from collected e-mail addresses*", present in Mozilla v1.6 and fixed in v1.7.
2. Bug #216154[7], described as "*Anchors in e-mails are broken - clicking anchor doesn't go to target in an e-mail*", existing in v1.5.1 and patched in v1.6.
3. Bug #225243[8], described as "*Text is printed in mirror-image on all pages in1.6a display and print-preview is correct, but the generated postscript is wrong*", present in v1.6 and fixed in v1.7.
4. Bug #209430[9], which is described as "*Ctrl+Delete and Ctrl+BackSpace delete words in the wrong direction*", present in v1.5.1 and fixed in v1.6.
5. Bug #231474[10], described as: "*If a user gets a lot of attachments (e.g. thirty) with only one short line the content of this file is damaged*", located in v1.5.1 and fixed in v1.6.

Methods and functions modified by developers to remove the bugs were identified by inspecting the patches endorsed by Mozilla source code reviewers. Table 5 reports the methods that have been patched.

As with Eclipse, more than one patch could have been produced in the process of fixing the bugs. In such a case, we considered the union of the changed methods from the first patch and the most recent patch.

Table 6 reports a narrative description of the two families of scenarios (one exercising the feature and the other not) used to obtain the SPR rankings and the queries used to rank methods using LSI.

We explain in more details the computation of the SPR rankings. Scenario 1, used to build the $C_{I'}$ was executed only once while we re-used all available scenario executions not interfering with the given bug detection to build C_I. This corresponds to mimicking a situation in which developers reuse as much as possible previous knowledge and collected data to accomplish theirs tasks.

For each bug, the scenarios produce events, identifying called methods and functions. These events are collected to build the $C_{I'}$ and C_I sets. Table 7 summarizes the numbers of unique methods and classes belonging to the $C_{I'}$ and C_I sets respectively for each bug.

$C_{I'}$ and C_I cardinalities are spread fairly evenly across bugs and even for the smaller sets, the task to manually identify methods responsible for the bug by comparing sets and inspecting methods or classes would be overwhelming. For example, bug #182192 involves one class and two methods while $C_{I'}$ and C_I sets contain thousands of classes and methods. Obviously, the methods to be modified must belong to either $C_{I'}$ or to the intersection $C_{I'} \cup C_I$, which contains respectively 188 and 2,694 unique methods, when compared using Unix *comm*. Yet, the cost of manually inspecting these 188 + 2,694 unique methods is still substantially higher than the effort required when studying E'_1 methods only. Table 8 reports E'_1 and E'_0 figures for Mozilla bugs. Clearly, the SPR technique takes

TABLE 5
METHODS AND FUNCTIONS RESPONSIBLE FOR THE UNWANTED FEATURES IN MOZILLA, GLOBAL FUNCTIONS ARE IDENTIFIED WITH THE 'ROOT::' PREFIX

Bug #	Methods
182192	nsAbAddressCollecter::CollectAddress nsAbAddressCollecter::CollectUnicodeAddress
216154	nsMailboxService::NewURI nsImapService::NewURI nsSmtpService::NewURI
225243	nsPostScriptObj::begin_page nsPostScriptObj::begin_document nsPostScriptObj:: setcolor nsPostScriptObj::colorimage nsPostScriptObj::grayimage nsRenderingContextPS::Init, nsRenderingContextPS::DrawScaledImage nsRenderingContextPS::LockDrowingSurface
209430	nsPlaintextEditor::DeleteSelection
231474	Root::MimeObject_parse_begin

[6] https://bugzilla.mozilla.org/long_list.cgi?buglist=182192
[7] https://bugzilla.mozilla.org/show_bug.cgi?id=216154
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=225243
[9] https://bugzilla.mozilla.org/show_bug.cgi?id=209430
[10] https://bugzilla.mozilla.org/show_bug.cgi?id=231474

TABLE 6
SCENARIOS AND QUERIES USED FOR THE MOZILLA

Bug #	Scenario 1 (Not exercising the bug)	Scenario 2 (Exercising the bug)	LSI Query
182192	A user replies to an e-mail	A user performs the same action as in Scenario 1. Then, using the mouse, the user collects the e-mail address of the sender. Plus any other scenario not related to the bug.	*collect collected sender recipient email name names address addresses addressbook*
216154	A user receives an e-mail with internal anchors and clicks on one anchor to access some text below in the document	A user receives an e-mail with internal anchors, reads and saves it. Plus any other scenario not related to the bug.	*mailbox uri url msg pop3 msgurl service*
225243	A user generates a postscript file by printing a page containing images.	A user accesses the same page but does not print it. Plus any other scenario not generating a postscript printing file.	*print page orientation portrait landscape postscript postscriptobj*
209430	A user accesses the Web page associated with a bug in Bugzilla and deletes words.	A user accesses the same page but does not delete words. Plus any other scenario which does not relate to the bug.	*Delete deleted word action*
231474	A user sends the sample e-mail with many attachments from Bugzilla gets this e-mail using the Mozilla mail reader.	A user reads e-mails without attachments. Plus any other scenario not related to the bug	*attachment encoding content mime*

advantage of its probabilistic nature and its ability to rank relevant versus non-relevant events. E'_1 and E'_0 cardinalities have to be compared with the corresponding cardinalities of $C_{I'}$, because at most E'_1 methods would require inspection if F' and F scenarios are carefully chosen. However, SPR cannot distinguish between two methods or classes involved in E'_1 sets.

Table 9 reports for each bug, the top ranked methods and their rankings with LSI, SPR, and PROMESIR, with $\lambda = 0.5$. For SPR, the ranking is the cardinality of the E'_1 set divided by two. We also compute the improvement factor that PROMESIR brings over the individual techniques. Again, PROMESIR increases dramatically the effectiveness of the feature location over the individual techniques.

We chose different values for λ but did not observe any significant changes in the final ranks of the methods when considering values of λ around 0.5. We further discuss the impact of λ on the final results in Section 4.7.

4.7. Discussion

Each feature location technique has its strengths and weaknesses. Given the different types of data and processes they use, they do not have common weaknesses and thus their combination overcomes in part their respective shortcommings. Each method relies on the developers as SPR requires a developer to define and run scenarios and LSI needs developer-defined queries. Poor choice of scenarios or queries impacts negatively the results. The SPR technique allows the developer to define multiple scenarios to address this problem, while the LSI-based technique provides the developer with terms related to her query to help improve it. The results of the LSI are impacted by the extent to which the comments and identifiers in the program reflect the domain or the developer's familiarity with the vocabulary of the program. These factors do not influence SPR, where the tendency is to rank more methods than necessary as relevant to a feature, while poor queries or language will return no relevant method in the top of the ranked list when using LSI.

The case studies provide data to assess the effectiveness of PROMESIR and to assess the help brought to the developer for identifying features in the source code. In the Mozilla case study the average improvement of

TABLE 7
CARDINALITYOF SETS $C_{I'}$ AND C_I MEASURED IN NUMBERS OF UNIQUE METHODS AND CLASSES FOR MOZILLA BUGS

Bug #	Unique Methods in $C_{I'}$	Unique Classes in $C_{I'}$	Unique Methods in C_I	Unique Classes in C_I	Unique Methods in $C_{I'} \cup C_I$	Unique Classes in $C_{I'} \cup C_I$
182192	8,693	2,057	12,030	2,306	12,252	2,325
216154	28,888	4,145	32,417	4,244	34,535	4,331
225243	19,446	2,774	30,551	4,570	40,051	4,667
209430	24,277	3,617	25,498	4,119	32,727	4,220
231474	18,870	2,864	34,073	4,268	35,858	4,347

TABLE 8
SIZE OF E'_T MEASURED IN METHODS AND CLASSES FOR MOZILLA BUGS FOR T=1 (100 % RELEVANT) AND T>0

Bug #	E'_1 methods	E'_1 classes	E'_0 methods	E'_0 classes
182192	188	72	230	82
216154	665	174	4,905	963
225243	944	164	3,958	873
209430	1,018	261	3,970	893
231474	988	234	3,524	688

PROMESIR over SPR was close to the one obtained in Ecplise (121 vs. 182), whereas the improvement in the LSI case is more dramatic (20 vs. 4). One possible explanation is the quality of the text in Eclipse with respect to the text in Mozilla. In addition, except for one bug in Mozilla, we ran only two queries for each bug, whereas in Eclipse we used three for two of the bugs and two queries for the other bug. Refining the queries more could have resulted in better results. In each case we found significant improvement in terms of effectiveness of PROMESIR over SPR or LSI alone.

The case studies support our claim that combining expert judgments improves the effectiveness of feature location. The results of the case studies show that the new technique performs better than any one of the two techniques alone.

In all case studies, we chose a value of $\lambda = 0.5$, this value allows PROMESIR to outperform LSI and SPR. We studied the change in the rankings of relevant methods as we varied λ and notice that $\lambda = 0.5$ is an adequate value, which means that neither technique is favoured over the other. In Figure 3, we present the rankings of the first relevant method for each bug (in Mozilla and in Eclipse) with respect to the value of λ, using a step of 0.01. This plot shows that as λ decreases the ranking of the first relevant method tends to be higher (i.e., the effectiveness of the hybrid technique deteriorates). This behaviour is expected because, as λ becomes close to 0.0 or 1.0, one expert is strongly favoured over the other (i.e., LSI over SPR respectively). In extremes, when λ is equal to 0.0, the ranking is provided by the LSI technique alone and by the SPR technique alone when λ is equal to 1.0 (see Table 4 and Table 9). With λ values between 0.2 and 0.99 we obtain quite stable results, similar to the ones obtained with $\lambda = 0.2$, showing that the two techniques truly complement each other.

As mentioned before, PROMESIR performs best when LSI and SPR are used in parallel, such that the result of each query is combined with SPR. In the case studies we combined only the rankings of the last query with SPR, which allowed a better comparison with LSI alone. PROMESIR would improve the results even more if the combination was iterative. For example, for one of the bugs (i.e., bug# 225243 from Mozilla) we refined the LSI query three times before we combined the results with SPR approach. In that case, the third LSI query ranked the first method related to the feature in position 24 and PROMESIR ranked it on position 6 (see Table 9). If we rank the results of the first LSI query with PROMESIR, then the same method ranks on position 12 (with LSI alone this method ranks 124).

In this work we do not compare PROMESIR with other techniques for feature location, because SPR and LSI have been compared with other approaches elsewhere: SPR was compared with *grep* and formal concept analysis applied on execution traces in [17], whereas LSI was compared with *grep* and search of dependencies in [28].

4.8. Threats to Validity

Several issues may have affected the results of the case studies and thus may possibly limit generalizations. We made all efforts to minimize the effect of these issues.

One such issue is the extent to which the programs used in the case studies are representative of those actually used in practice. Although Eclipse and Mozilla are real-world programs, this threat could be reduced if we experiment with other programs of different sizes and domains.

Another issue is the use of scenarios to obtain rankings of methods with SPR, because we could have chosen by chance "best" or "worst" scenarios to identify the features. However, the results of the case studies are posi-

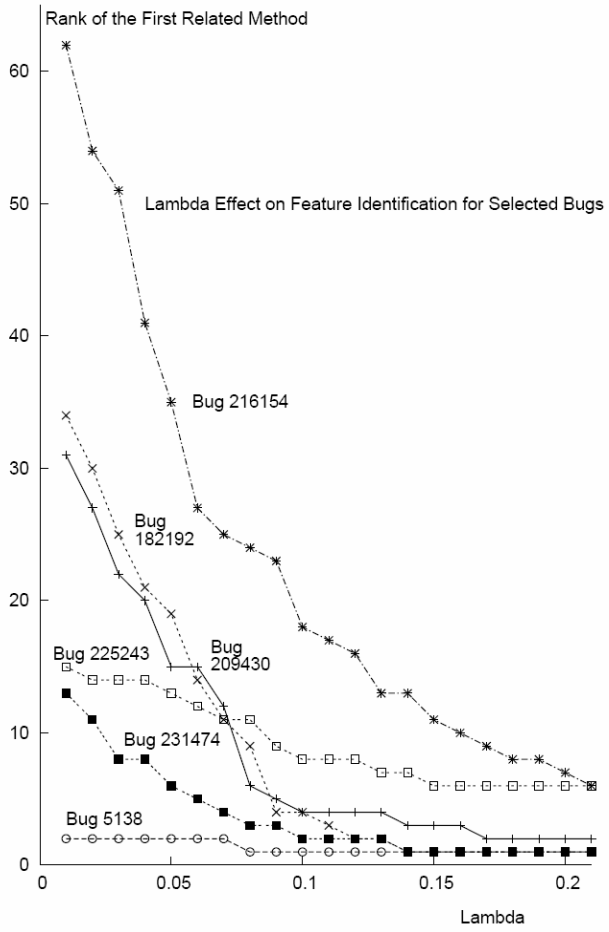

Fig. 3. The effect of λ on the PROMESIR ranking (each line shows the effect of λ on the rankings of the first method related to the considered bug)

TABLE 9
EFFECTIVENESS OF EACH TECHNIQUE FOR THE MOZILLA BUGS

Bug #	E_{LSI}	$E_{PROMESIR}$	E_{SPR}	Method name	PROMESIR over LSI	PROMESIR over SPR
182192	37	2	94	nsAbAddressCollecter:: CollectAddress	18×	47×
216154	76	6	332	NsMailboxService::NewURI	12×	55×
225243	24	6	472	nsPostScriptObj::begin_page	4×	80×
209430	49	1	509	nsPlaintextEditor::DeleteSelection	49×	509×
231474	18	1	494	ROOT::MimeObject_parse_begin	18×	494×

tive although we are not experts in the Eclipse or Mozilla source code and cannot ensure that our scenarios are the best to capture those particular features. By presenting several results, we expect that the used scenarios are averaged in the sense of their ability to properly capture the desired features.

The queries formulated to obtain LSI rankings are dependent on the developer's knowledge, thus the results may be impacted by the actual query. However, as we discussed in several of the examples, the developer does not need an extensive knowledge of the source code to formulate LSI queries, which with PROMESIR will produce good results. In fact, even in the naïve situation, where a developer uses the bug description as a query, PROMESIR performs better than LSI alone.

The effectiveness measure for SPR is defined on an average case scenario. In reality, a developer may find the relevant method in a set indicated by SPR much faster. Nonetheless, given the large difference between the PROMESIR and SPR accuracies, modifying this assumption would not change the results dramatically.

We could take into account the number of unsuccessful queries when applying LSI. However, in most cases, the results investigated after the first query would be in the top of the results for the second query, so the developer does not need to investigate these results again and thus there is no real increase in the searching effort. Also, it could be possible to penalize the effectiveness measure with a value depending on the number of queries. Still, this will not affect the overall results as the new measure will change the PROMESIR rankings as well.

The features may be implemented by more methods than those suggested by a patch, as correcting the problem may involve just part of the implementation. This fact does not influence our results because considering more relevant methods increases or maintains the same effectiveness of the techniques.

In addition, we ran LSI and SBP in parallel, thus penalizing PROMESIR. Using PROMESIR in an iterative fashion, as described in Section 3, yields better accuracy, as mentioned in Section 4.7.

5. CONCLUSIONS AND FUTURE WORK

The main contribution of the paper is a new technique (PROMESIR) for feature location that combines an information retrieval technique (LSI) with a dynamic technique (SPR). We used PROMESIR in a set of case studies for bug location in two large open-source programs, Eclipse and Mozilla. The case studies showed that LSI and SPR, based on different analysis methods and data, complement each other, and the results obtained with the combined techniques are better than those of any one of the techniques used independently.

Feature location using PROMESIR proves to be accurate and fast, but the technology behind is computationally intensive. We will focus some of our future efforts on making the supporting tools as effective as possible. In addition, we plan to investigate how well PROMESIR can be used to support tasks subsequent to feature location, such as impact analysis and change propagation, when the entire feature implementation needs to identified.

ACKNOWLEDGMENT

Giuliano Antoniol was partially supported by NSERC, Canada Research Chair in Software Change and Evolution. Yann-Gaël Guéhéneuc was partially supported by NSERC, Discovery Grant. Both Andrian Marcus and Václav Rajlich were partially supported in this work through grants from the National Science Foundation (CCF-0438970) and the National Institute for Health (NHGRI 1R01HG003491) and 2006 IBM Eclipse Innovation Awards.

REFERENCES

[1] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension," *Proc. IEEE International Workshop on Program Comprehension (IWPC'02)*, 2002, pp. 271-278.

[2] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," in *IEEE Software*, 2004, pp. 2-9.

[3] S. Bohner and R. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

[4] J.-P. Queille, J.-F. Voidrot, N. Wilde, M. Munro, and "The Impact Analysis Task in Software Maintenance: A Model and a Case Study," *Proc. International Conference on Software Maintenance*, 1994, pp. 234 - 242.

[5] L. C. Briand, J. Wuest, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *Proc. International Conference on Software Maintenance*, 1999, pp. 475-483.

[6] M. Lindvall and K. Sandahl, "Traceability Aspects of Impact Analysis in Object-oriented Systems," *Journal of Software Maintenance: Research and Practice,* vol. 10, no. 1, pp. 37-57, 1998.

[7] G. Antoniol, G. Canfora, G. Casazza, and A. Lucia, "Identifying the Starting Impact Set of a Maintenance Request: A Case Study," *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR2000)*, pp. 227-231.

[8] V. Rajlich, "Changing the Paradigm of Software Engineering," in *Communications of ACM*. vol. August, 2006, pp. 67-70.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Transactions on Software Engineering,* vol. 29, no. 3, pp. 210 - 224, March 2003.

[10] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Towards Employing Use-Cases and Dynamic Analysis to Comprehend Mozilla," *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 639-642.

[11] D. Eng, "Combining static and dynamic data in code visualization," *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'02)*, 2002, pp. 43-50.

[12] G. Antoniol and Y. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study," *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 357-366.

[13] M. Ernst, "Static and Dynamic Analysis: Synergy and Duality," *Proc. ICSE Workshop on Dynamic Analysis (WODA'03)*, 2003, pp. 24-27.

[14] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," Software Engineering Research Center 2004.

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

[16] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214-223.

[17] G. Antoniol and Y. G. Guéhéneuc, "Feature Identification: An Epidemiological Metaphor," *IEEE Transactions on Software Engineering,* vol. 32, no. 9, pp. 627-641, 2006.

[18] R. Jacobs, "Methods for combining experts' probability assessments," *Neural Computation,* vol. 7, no. 5, pp. 867-888, September 1995.

[19] R. Winkler and R. Clemen, "Multiple Experts vs. Multiple Methods: Combining Correlation Assessments," *Decision Analysis,* vol. 1, no. 3, pp. 167-176, September 2004.

[20] N. Wilde and T. Gust, "Locating User Functionality in Old Code," *Proc. IEEE International Conference on Software Maintenance*, 1992, pp. 200-205.

[21] A. D. Eisenberg and K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 337-346.

[22] E. Wong, W. Gokhale, S. S., and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software,* vol. 54, no. 2, pp. 87-98, Oct. 2000.

[23] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program Understanding and the Concept Assignment Problem," *CACM,* vol. 37, no. 5, pp. 72-82, May 1994.

[24] A. V. Aho, "Pattern matching in strings," in *Formal Language Theory: Perspectives and Open Problems* New York: Academic Press, 1980, pp. 325-347.

[25] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," *Proc. 8th IEEE International Workshop on Program Comprehension (IWPC'00)*, 2000, pp. 241-249.

[26] M. Robillard, "Automatic Generation of Suggestions for Program Investigation," *Proc. Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 11 - 20

[27] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Non-interactive Approach to Feature Location," *ACM Transactions on Software Engineering and Methodologies (TOSEM),* vol. 15, no. 2, pp. 195-226, 2006.

[28] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code," *Proc. 13th IEEE International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 33-42.

[29] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: from object-interactions to feature-interactions," *Proc. 20th IEEE International Conference on Software Maintenance (ICSM'04)*, 2004, pp. 72-81.

[30] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of Traceability Links Between Software Documentation and Source Code," *International Journal of Software Engineering and Knowledge Engineering,* vol. 15, no. 4, pp. 811-836, October 2005.

[31] L. H. Etzkorn and C. G. Davis, "Automatically Identifying Reusable OO Legacy Code," *IEEE Computer,* vol. 30, no. 10, pp. 66-72, October 1997.

[32] W. Frakes and B. A. Nejmeh, "Software Reuse through Information Retrieval," *ACM SIGIR Forum,* vol. 21, no. 1-2, pp. 30 - 36 Sept.-March 1986.

[33] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering,* vol. 17, no. 8, pp. 800-813, 1991.

[34] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," *Proc. IEEE/ACM International Conference on Software Engineering (ICSE'02)*, 2002, pp. 513-523.

[35] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," *Proc. 23rd International Conference on Software Engineering (ICSE'01)*, Canada, 2001, pp. 103-112.

[36] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," *Proc. Automated Software Engineering (ASE'01)*, 2001, pp. 107-114.

[37] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Enhancing an Artefact Management System with Traceability Recovery Features," *Proc. 20th IEEE International Conference on Software Maintenance (ICSM'04)*, 2004, pp. 306-315.

[38] D. Poshyvanyk and A. Marcus, "The Conceptual Coupling Metrics for Object-Oriented Systems," *Proc. 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, 2006, pp. 469 - 478.

[39] A. Marcus and D. Poshyvanyk, "The Conceptual Cohesion of Classes," *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 133-142.

[40] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," *Proc. 12th Working Conference on Reverse Engineering*, 2005, pp. 133-142.

[41] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE Transactions on Software Engineering,* vol. 32, no. 1, pp. 4-19, January 2006 2006.

[42] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "ADAMS Re-Trace: a Traceability Recovery Tool," *Proc. Proceedings of the*

Ninth European Conference on Software Maintenance and Reengineering (CSMR'05), 2005.

[43] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev, "IRiSS - A Source Code Exploration Tool," *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 69-72.

[44] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - an Eclipse plug-in for Source Code Exploration," *Proc. 14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 252-255.

[45] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering (TSE)* vol. 32, no. 12, pp. 971-987, December 2006.

[46] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 12, pp. 889- 903, 2004.

[47] R. K. Yin, *Applications of Case Study Research*, 2 ed. CA, USA: Sage Publications, Inc, 2003.

[48] Y. G. Guéhéneuc and A. A. Hervé, "Recovering Binary Class Relationships: Putting Icing on the UML Cake," *Proc. 19th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'04)*, 2004, pp. 301--314.

[49] D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," *Proc. 14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 137-146.

Denys Poshyvanyk is a PhD candidate in the Department of Computer Science at Wayne State University. He received MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006 respectively. His research interests include software maintenance and evolution, program comprehension, program analysis, software metrics, and software visualization. He is a Microsoft Student Partner for Wayne State University. He is member of the IEEE and ACM.

Yann-Gaël Guéhéneuc is assistant professor at the Department of Computing Science and Operations Research of University of Montreal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He is interested also in empirical software engineering where he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals.

Andrian Marcus is assistant professor in the Department of Computer Science at Wayne State University. He received his PhD in Computer Science from Kent State University in 2003 and he has prior degrees from the University of Memphis and the "Babeş-Bolyai" University of Cluj, Romania. His research interests include software evolution, program understanding, and software visualization, focusing on using information retrieval techniques to support software engineering tasks. He serves on the Steering Committee of the IEEE International Conference on Software Maintenance since 2005. He is recipient of a Fulbright Junior Research Fellowship in 1997.

Giuliano Antoniol received his degree in electronic engineering from the Universita' di Padova in 1982. In 2004 he received his PhD in Electrical Engineering at the École Polytechnique de Montréal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research Chair Tier I in Software Change and Evolution. Giuliano Antoniol published more than 100 papers in journals and international conferences. He served as a member of the Program Committee of international conferences and workshops such as the International Conference on Software Maintenance, the International Conference on Program Comprehension, the International Symposium on Software Metrics. He is presently a member of the Editorial Board of the Journal Software Testing Verification & Reliability, the Journal Information and Software Technology, the Journal of Empirical Software Engineering and the Journal of Software Quality. He is currently Associate Professor in the École Polytechnique de Montréal, where he works in the area of software evolution, software traceability, software quality and maintenance.

Václav Rajlich is a professor and former chair of the Department of Computer Science at Wayne State University. His research interests include program comprehension and program evolution. He published approximately 80 peer-reviewed articles.