# Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace

Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202
1-313-577-5408
[dliu, amarcus, denys, rajlich]@wayne.edu

## ABSTRACT
The paper presents a semi-automated technique for feature location in source code. The technique is based on combining information from two different sources: an execution trace, on one hand and the comments and identifiers from the source code, on the other hand.

Users execute a single partial scenario, which exercises the desired feature and all executed methods are identified based on the collected trace. The source code is indexed using Latent Semantic Indexing, an Information Retrieval method, which allows users to write queries relevant to the desired feature and rank all the executed methods based on their textual similarity to the query.

Two case studies on open source software (JEdit and Eclipse) indicate that the new technique has high accuracy, comparable with previously published approaches and it is easy to use as it considerably simplifies the dynamic analysis.

## Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – enhancement, restructuring, reverse engineering, and reengineering

## General Terms
Algorithms, Performance, Design, Experimentation

## Keywords
Program understanding, feature identification, concept location, dynamic and static analyses, information retrieval

## 1. INTRODUCTION
Identifying the parts of the source code that correspond to a specific functional requirement is one of the most common and important activities undertaken by software engineers during software evolution. This activity is known as *feature* or *concept location* [36]. The goal of feature or concept location is to identify some part of the source code, for example a single method, which will be modified in response to a change request and in this way it gives the programmer a starting point in this process. The full extent of the change is later defined through the separate activities of impact analysis and change propagation [26, 27].

The main difference between concepts and features is that the user can exercise the latter. Features are usually described in the requirements of the software system. There are other concepts, usually from the solution domain, that do not necessarily correspond to features, such as a "linked list" or a "hash table" and hence the notion of concept is more general than the notion of feature. This paper deals with feature location only.

While in small systems, developers can perform feature location manually, it is more often than not that tool support is necessary, especially for large and complex software systems. Tool support for feature location was addressed in previous work [3, 25, 36, 38]. Depending on how such tools extract information from the source code, there are two flavors of (semi)automated feature location techniques: static and dynamic. If the information is gleaned without executing the subject program, both the information and the tool are categorized as static; otherwise they are dynamic. Dynamic techniques are based on collecting and analyzing execution traces and mapping them to the source code [3, 9, 12, 33, 35, 36, 38]. Static techniques use program dependencies and the textual information from the source code and associated documentation to help the user search the software [1, 4, 29, 32, 40]. A number of techniques use both types of analyses [11, 25].

Both static and dynamic techniques have their own limitations. In general, dynamic techniques are conservative in nature, as execution traces are often very large and contain a lot of noise, as stated in [3] "we cannot distinguish feature-relevant and feature-irrelevant events with one unique trace alone. We need multiple traces from different scenarios and exercising different features to identify feature-relevant events". Selection of proper test cases or scenarios to be executed is another problem of these techniques. Simmons et al [33] reckon that "poorly chosen test cases that exercise too much or too little of the system may cause problems". Most dynamic techniques use at least two execution traces, where the role of one is to filter the other. Complex mechanisms were proposed to improve the trace filtering problem (see Section 4 for details). The hybrid techniques usually aim at


*ASE'07*, November 5-9, 2007, Atlanta, Georgia, USA.

the same problem, where static information is used to filter the execution traces. Previous work [25] provided evidence that the information obtained from overlapping traces, on one hand, and textual information obtained from the source code, on the other hand, are orthogonal with respect to feature identification, thus their combination results in a very effective feature location technique.

In this paper we introduce a novel hybrid feature location technique. The proposed approach is based on the idea that a single execution trace of a scenario, exercising a feature of interest, contains *all* the necessary information to find the most important parts of the source code that are implementing this feature [12, 15, 17, 30] and that filtering the single trace with data based on textual analysis is sufficient to extract the relevant pieces of source code. More than that, developers can construct marked-traces [30] to reduce the size of the traces. Filtering is done using Latent Semantic Indexing (LSI) [7], an Information Retrieval (IR) method. LSI is used to index the textual information from the source code (that is, comments and identifiers) and allows the users to run queries, describing a feature in natural language and obtain results as a ranked list of source code elements (that is, classes, methods, or functions). The novel feature location technique is named **SI**ngle **T**race and **I**nformation **R**etrieval (SITIR).

The next section describes SITIR and its component technologies. Section 3 presents two case studies that emphasize the effectiveness of SITIR when compared to the results obtained with earlier similar approaches. Section 4 presents other related dynamic feature location approaches and discusses their differences and similarities to SITIR. Section 5 concludes the paper and shows future research directions.

# 2. THE PROPOSED APPROACH

In order to present the details of SITIR, we need to discuss how the dynamic analysis is performed in order to obtain the execution traces. We also present briefly how LSI is used for feature location in this work.

## 2.1 Dynamic Analysis

In this work, the dynamic analysis uses a profiling tool, namely Java Platform Debugger Architecture (JPDA)[1]. Although other approaches for collecting execution traces are available, for example, source code and byte-code instrumentation, or even instrumentation of the Java virtual machine [34], we opted for the JPDA-based approach to obtain execution traces, because it offers flexibility and ease of use. The JPDA-based tracer allows the user to collect marked-traces [30] by manually controlling when to start and stop tracing. It organizes tracing information into separate thread-based log files and provides support for method- and class-level granularity as well as multiple output formats for the execution traces. For feature location, the tracer outputs a set of methods (or classes) that were executed in each thread.

Internally, JPDA has three layers: the Java Virtual Machine Tool Interface (JVMTI, which substituted JVMPI and JVMDI in Java 5.0), which works in Java virtual machines; the Java Debug Wire Protocol (JDWP), which serves as a standard communication protocol; and the Java Debug Interface (JDI), which supplies the programmer with a high-level Java language interface.

We use JPDA for tracing as follows. The tracer starts up a separate Java Virtual Machine (JVM) on which the subject program is executed. Once the tracing procedure is initiated, the tracer sends a message to the second JVM specifying what kind of events it should report and what packages/classes it should ignore. The tracer can change its settings any time by sending different instructions to the second JVM. Thus, the tracer communicates to the second JVM only, but not to the subject program. The subject program runs on the second JVM and interacts with the running environment without any knowledge of the tracing utility. In such a way, the interference of the tracing tool with the subject program is minimal (from the program's point of view it seems like it is running on a slower machine). The fact that the tracer is running on the first JVM is transparent to the environment.

Other researchers used JPDA for collecting execution traces. Salah et al. [30] used JVPROF, which is built on top of JPDA, to record the method call sequences for different scenarios. Reiss and Renieris [28] used TMon, which is built on top of JVMPI to trace Java programs.

## 2.2 Information Retrieval based Ranking of Methods

Using advanced IR techniques, such as LSI, allows users to capture relations between terms (words) and documents in large bodies of text. A significant amount of domain knowledge is embedded in the comments and identifiers present in source code. Using IR methods, users are able to index and effectively search this textual data formulating natural language queries, which describe the concepts they are interested in. Identifiers and comments present in the source code of a software system form a language of their own without a grammar or morphological rules. LSI derives the meanings of words from their usage in passages, rather than a predefined dictionary, which is an advantage over existing techniques for text analysis that are based on natural language processing [32].

In software engineering, LSI has been used for a variety of tasks closely related to feature location, such as software reuse [14, 19, 39], abstract data types identification [20], high level concept clone detection [21], traceability link recovery between software artifacts [2, 6, 22], topic identification in source code [18], requirements tracing [16], etc.

We introduced previously [24] a methodology to index and search the source code using LSI. The methodology was subsequently refined and combined with dynamic information (see Section 4 for details) to improve its effectiveness [25].

In a nutshell, the comments and identifiers from the source code are extracted and a corpus is created, where each document corresponds to a method in the system. LSI indexes this corpus and creates a signature for each document (method). These indices are used to define similarity measures between methods. As LSI does not use a predefined grammar or vocabulary it is very robust with respect to outlandish identifier names and stop words (which are subsequently eliminated). Users can originate queries in natural language (as opposed to regular expressions or some other structured format) and the system returns a list of all the methods in the system, ranked by their semantic similarity to

[1] http://java.sun.com/javase/technologies/core/toolsapis/jpda/

the query. The use is similar to many existing web search engines.

As the use of LSI in this work is similar to its previous uses, we refer the interested reader for more details on this approach and on LSI to [23-25].

## 2.3 Single Execution Traces and Information Retrieval

SITIR is a semi-automated technique, which implies that the user input is needed and of course, results are sensitive to that input. Developers have to decide on a scenario that will exercise the desired feature. Using the tracing utility, a marked-trace is obtained, from which a set of uniquely executed methods is extracted. More precise marking will generate more compact traces. The user is involved in the marking as well.

The user then formulates a query as a set of terms (such as, words or identifiers) describing the feature. The more knowledge the user has about the system, the better the query (and its results) will be. The set of uniquely executed methods is sorted based on their semantic similarity to the user query, computed with LSI.

Prior to using SITIR for feature location the tracing tool should be configured and the software system needs to be indexed with LSI. The indexing is a one time process and only needs to be redone if significant changes are done to the source code. This process requires minimum user involvement.

The feature identification methodology with SITIR requires little domain or software system specific knowledge and it consists of the following steps:

1. **Formulating and executing a single scenario.** The developer formulates a *scenario* that captures the feature of interest; she marks the intervals in this scenario for which the trace should be collected and runs the scenario. A set of executed methods is obtained. If the user is uncertain on where to mark the traces, complete scenarios can be executed.
2. **Formulating the query.** The developer selects a set of terms, a *query*, which describe the feature. The tool checks whether the words from the query are present in the vocabulary of the source code (produced by LSI). If some word is not present, then the tool suggests similar words or it eliminates the word from the initial query.
3. **Ranking the executed methods.** Based on the LSI index, the set of methods generated in step 1 is sorted based on the similarity between the methods and the user query. The ranked list of methods is presented to the user.
4. **Examining the results.** The programmer inspects the methods ranked in step 3, starting with the methods on the top of the list. For every method in the list, the developer makes a decision whether the method belongs to the feature or not. If it is part of the feature, then SITIR stops. If it is not and new knowledge obtained from the investigated documents helps to reformulate the scenario or to write a better query, then the user is directed back to step 1 or 2, as needed. The user may opt to reformulate the scenario, the query, or both.

The feature location process based on SITIR is interactive, but the user's role is relatively simple. In our previous experience [23, 25] most users tend to look at less than ten methods before interacting with the system to improve the results. The goal of SITIR is to rank relevant methods within the top ten. Given the complementary nature of the analyses employed in SITIR (that is, textual and dynamic), the user can improve the results by either reformulating the query or reducing the part of scenario which will be executed via marking mechanism. The case studies presented in the next section show that the SITIR is more sensitive to query reformulation than trace size reduction, however combining the results of the analysis of textual information and the analysis of the execution traces produces significantly better results than any of these techniques if used on a standalone basis.

# 3. CASE STUDIES

In order to evaluate the performance of SITIR, we designed and conducted several case studies. Our assumption was that SITIR performs better than LSI used alone and better than using the single execution trace alone.

We present here two different case studies. In the first case study we used SITIR to locate three features in JEdit[2] associated with change requests. In the second case study, we replicated a previously published case study for locating three features associated with Eclipse[3] bugs [25]. In the second case study, we were able to compare the results obtained by SITIR with two other approaches, namely Probabilistic Ranking Of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) [25] and Scenario-based Probabilistic Ranking (SPR) [3].

## 3.1 Design and objectives of the case studies

In these case studies we chose methods as the level of document granularity. In other words, SITIR returns to the user a set of ranked methods for investigation. Note that SITIR can be used also with a class level granularity, where classes are returned to the user. In order to compare SITIR with other techniques, we assess the effectiveness of each feature locating techniques by considering the ranking of a first method implementing the feature, which is relevant to the change request. We consider a method relevant to a change request if it will be modified in response to it. All the features investigated in these case studies are linked to explicit change requests or implicit ones (i.e., bug descriptions). Clearly not all methods implementing a feature are relevant to specific change requests. For example, most features have a corresponding part of the GUI, which is usually easy to find, but often does not change. Let's assume the "print" feature of a text editor program. It is likely that most such editors have a menu item labeled "print". If a change request states that we need to add a new feature that prints the selected text (not a usual feature in most editors), one will have to locate relevant methods that implement the "print" feature and methods that implement the "select text" feature. We are considering methods relevant to a change request, as their identification can be matched against available changes, thus providing an objective mechanism for evaluation.

The goal of each feature location technique is to reduce the programmer's effort in finding such methods. Once such a method is identified, other methods relevant to the change and to

[2] http://www.jedit.org/
[3] http://www.eclipse.org

the feature are inferred by following program dependencies [5] or inspecting the history of common changes [41], etc. So, a feature location technique is considered better than another one if it returns at least one method relevant to the feature on a better position in the list of ranked results. In such a situation, the user needs to inspect fewer methods before she finds the relevant one.

We chose one large and one medium-size open-source systems to show the scalability of our novel technique and to allow replication of our case studies.

One of the goals of the presented case studies is to allow for quantitative evaluation between different techniques. This is a notoriously difficult task (see the related work section) as it is hard to define the entire extent of the implementation of a feature in large systems. One feature may be implemented by hundreds of methods and many of them may contribute towards several features. In order to have a gold standard against which we can define objective measures, we narrow the extent of feature implementation to those methods relevant to a change request.

For the first case study on JEdit, we located the starting point for three distinct features, originally described in two change requests. These features were implemented during a graduate class project by students in our research lab, none of which co-authored this paper. We used their implementations to verify the correctness of the results produced by SITIR. In this case study we also studied the extent to which query refinement and selection of scenarios impacts the results returned by SITIR.

In the second case study on Eclipse, we use documented bugs to asses our method and compare that with related approaches, as reported in [25]. The bug description is considered as a change request.

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 that bug. We consider these changed methods as belonging to the feature associated with the bug. One method may belong to more than one bug (that is, changed in different bug fixes), but it is at least exercised in the associated feature. We do not attempt to identify defects (that is, the root cause of a bug) such as a condition for infinite loop because SITIR works at the method level and therefore, no information on the executed statements is available. In order to allow replication of the results, we located the same bugs as those studied in [25]. The following major criteria were previously used to select those bugs:

- The bugs should be well-known, documented, and reproducible;
- The bugs should have available and approved patches applied in recent releases.

## 3.2 Objects of the Case Studies

JEdit 4.2 is an open source programmer's text editor, which consists of approximately 500 classes implemented in about 5,000 methods with about 88,000 lines of Java source code and internal comments.

Eclipse is an open-source integrated development environment (IDE) used both in the open-source and industrial development settings. Eclipse is mostly written in Java, with some C/C++ code used mainly for the widget toolkit, which we did not analyze within this case study. We used version 2.1.3, which contains approximately 7,000 classes with about 89,000 methods in more than 8,000 source code files implemented in nearly 2.4 MLOC.

In both case studies we followed the approach for indexing and ranking the source code methods with LSI briefly outlined in Section 2.2. We built the corpus for Eclipse and JEdit by extracting all comments and identifiers from all Java source code files in these systems. The extracted source code is processed as follows: predefined tokens are eliminated (such as, operators, special symbols, numbers, Java programming language keywords, Java standard library class names, etc.); the identifier names in the source code are split based on the naming conventions observed in Eclipse and JEdit while the original form of every identifier is kept as well (for details on this procedure refer to the previous work in [22, 25]); every document in the corpus is created with the comments and identifiers corresponding to each method in every software system. We used a dimensionality reduction factor of 500, as in our previous case studies [25], which adequately represents a semantic space of large programs such as Eclipse. The Eclipse corpus has 56,861 unique terms encountered in approximately 89,000 documents (methods) and the JEdit corpus has 7,353 distinctive terms in about 5,000 documents.

## 3.3 Evaluation

In order to evaluate SITIR, we need a standardized measure to compare with it other feature location techniques. We used the *effectiveness measure*, introduced in [25], since it allows comparing SITIR directly with other techniques such as PROMESIR and SPR. We decided not to use standard Information Retrieval measures such as precision and recall because SITIR will rank *all* the executed methods. Thus, without a threshold, recall will always be 100%, whereas precision will be 1/n, where n is the number of executed methods to a given scenario. We could potentially utilize some configurable threshold based on initial observations of SITIR performance however such a solution would artificially increase the complexity of the novel technique, whereas our major goal is to keep that simple with as little overhead for the user as possible.

The rank of the first changed method related to the feature of interest is used to define the effectiveness measure. Since the objective of every feature location technique is to reduce the developer's effort during the location process, we measure this effort as the number of methods which appear in the final ranked list that the developer needs to investigate. The effectiveness measure of a technique $i$, $E_i$ is defined as the rank $r(m_i)$ of the method $m_i$, where $m_i$ is the top-ranked method according to the gold standard among the methods that must be changed (that is, implementing a part of the located feature, which is relevant to the change request). A lower value for $E_i$ indicates less effort, hence a more effective technique.

## 3.4 Locating features in JEdit

The features we locate in the JEdit case study are based on two change requests:

1. Add a "*Search and mark all*" menu item in the "*Search"* menu, which will locate all matches to a search phrase and add markers to all of the lines.
2. Currently jEdit shows a red dot at the end of every line. *Newline* is the only whitespace symbol that jEdit shows. Add a menu item "*Show/Hide whitespace*" under menu

**Table 1. Scenarios (with marks) and queries for locating three features in JEdit**

| Feature | Precise scenarios that exercise features | LSI queries: original +(refined) |
|---|---|---|
| **Search** | The programmer opens a document, opens the Search and Replace dialog by selecting the menu item Search \| Find, puts into the search textbox a word which exists in the current document, ***starts tracing***, clicks the button Find, waits until a matched place is highlighted, and then ***stops tracing***. | search find + (next) |
| **Add marker** | The programmer opens a document, ***starts tracing***, selects the menu item Markers \| Add/Remove Marker, waits until a marker is shown at the beginning of the text line, and then ***stops tracing***. | add marker |
| **Show whitespace** | The programmer opens a document with whitespace hidden, ***starts tracing***, selects the menu item View \| Show/Hide Whitespace, waits until whitespace is shown, and then ***stops tracing*** | whitespace text area visible + (paint) |

"*View*" to allow the user to choose whether all whitespace symbols (newlines, blanks, and tabs) will be shown. At this stage you do not have to worry about editing of the text with whitespace showing.

From the first change request we extracted two distinct features affected by this change (#1 and #2 below) and from the second change request we extracted one feature (#3 bellow):

1. "Search": searching for the occurrence of the provided search phrase.
2. "Add marker": adding a marker to the selected line in the text.
3. "Show whitespace": showing whitespaces as a symbol in the text.

In order to locate these features with SITIR, the scenarios described in Table 1 have been defined and executed to obtain the marked-traces. One of the co-authors of the paper with good knowledge of JEdit formulated the scenarios. We also executed one query for each feature, shown in Table 1 using LSI. Another one of the co-authors, also with good knowledge of JEdit, formulated the queries, independently of the previous co-author. The first relevant methods encountered in the search results for each feature are as follows:

- jedit.search.SearchAndReplace.find for the "search" feature;
- jedit.Buffer.addMarker for the "add marker" feature; and
- edit.textarea.TextAreaPainter.paintValidLine for the "show whitespace" feature.

The effectiveness measure is computed based on the rank of these methods (see Table 2) for each feature.

Given these scenarios, the tracing utility generated traces with the following number of unique methods:

- 202 methods for the "search" feature;
- 304 methods for the "add marker" feature, and
- 284 methods for the "show whitespace" feature.

The list of executed methods extracted from an execution trace is not ranked. One could rank them based on the order of their first call. Given that such an ordering is not really related to the change request, we do not define the effectiveness measure for a single trace.

For the "add marker" feature the relevant method topped the ranked list (see Table 2). For the "search" and "show whitespace" features the top relevant methods are ranked on positions 14 and 7 respectively by SITIR. In these two cases we refined each query by adding an additional term to each query, shown in parentheses in the last column of Table 2. The LSI results and the SITIR results were affected, as reflected by the number in the parentheses in Table 2. The LSI ranking improved from 59 to 36, and 56 to 43 respectively, while the SITIR ranks improved from 14 to 9 and from 7 to 5, respectively. These results show the effect of refining the user queries on the effectiveness of LSI and SITIR.

As discussed earlier, another way to improve the results is to reduce the size of the traces by using marked-traces. We investigated the effect of marking the scenarios on the effectiveness of the results produced by SITIR. We attempted to locate the same features without selecting marked traces, but rather automatically recording the execution of the complete scenario, including launching and closing JEdit. Such a usage is likely when programmers know little about how the software under analysis behaves and which features can be captured by particular scenarios.

**Table 2. Effectiveness of each technique for locating features in JEdit. The results for refined queries are in parentheses**

| Feature | $E_{LSI}$ | $E_{SITIR}$ |
|---|---|---|
| **Search** | 59 (36) | 14 (9) |
| **Add marker** | 5 | 1 |
| **Show whitespace** | 56 (43) | 7 (5) |

As expected, the resulting traces increased in size (for "search" we collected 1,477 unique methods; for "add marker" – 1,478 unique methods, and for "show whitespace" – 1,462 unique methods), however we did not observe significant drop of the rankings of the first relevant method in the SITIR results (see Table 3). Table 3 provides the results (that is, the ranks of the best ranked relevant methods) for all possible combinations of original and refined queries with methods captured with full and marked-traces for the "show whitespace" and "search" features.

**Table 4. Results for different queries produced by four developers for each of the features in the case study (the developer ID is indicated in the column DevID)**

| Feature | DevID | Query | LSI | SITIR (marked) | SITIR (full) |
|---|---|---|---|---|---|
| Search | 1 | search find phrase word text | 61 | 6 | 11 |
| | 2 | search final all forward backward case sensitive | 243 | 20 | 57 |
| | 3 | find search locate match indexof findnext | 32 | 6 | 13 |
| | 4 | searchdialog find findbtn searchselection save searchfileset searchandreplace | 189 | 11 | 36 |
| Add marker | 1 | marker select word display text | 26 | 1 | 5 |
| | 2 | add remove marker markers | 1 | 1 | 1 |
| | 3 | select highlight mark change background | 3242 | 160 | 662 |
| | 4 | buffer addmarker marker selection | 20 | 4 | 5 |
| Show whitespace | 1 | red dot newline whitespace view show display tab | 956 | 30 | 152 |
| | 2 | show hide whitespace blank space display | 626 | 48 | 130 |
| | 3 | symbol replace changecolor setvisible addlayer whitespace loadsymbol | 497 | 16 | 104 |
| | 4 | userinput textareapainter paint whitespace newline pnt | 78 | 8 | 23 |

Since the method relevant for "add marker" was already ranked on top, we did not change any parameters for that case.

**Table 3. Results for different combinations of original and refined queries with full and marked-traces while locating the "show whitespace" and "search" features in JEdit with SITIR**

| Feature | Trace | Original query | Refined query |
|---|---|---|---|
| Search | Full | 23 | 23 |
| | Marked | 14 | 9 |
| Show whitespace | Full | 25 | 25 |
| | Marked | 7 | 5 |

Based on our previous experience with using LSI for concept location [24, 25] we assumed that the effectiveness of SITIR would be most sensitive to the user queries. Users with little knowledge of the software system or its domain will write poor queries, while users with good knowledge of the software system will write better ones.

In order to assess this sensitivity to user queries we ran several queries formulated by several programmers based on the rules described in Section 2.3. We asked four members of our research lab to formulate queries which best describes each feature. These queries and the results of LSI and SITIR are presented in Table 4. The results of SITIR are obtained by using rankings produced via the given queries to filter the same execution traces in each case (for every feature we had a marked and full execution traces). The users had different levels of experience with JEdit: the first one (labelled with #1 in Table 4) never used JEdit, nor has he ever seen its source code, but he is familiar with the application domain. The other users had some knowledge of the JEdit source code as they implemented other features for JEdit, which did not relate to those used in this case study.

We observe that SITIR significantly reduces the search space even in those cases when users formulate peculiar queries, given the fact that they are unfamiliar with the system vocabulary and thus produce low LSI ranks. This data does not support our initial assumption and supports the fact that users do not have to formulate precise queries to capture the feature of interest as this query will be used to rank an already reduced search space of methods, which are executed with respect to given scenario. The data supports the idea that LSI and the execution traces each capture different information about the implementation of features in the source code, information which is complementary to each other.

## 3.5 Locating features in Eclipse

We applied SITIR to locate three different features in Eclipse. In this case study, the features are associated with bug descriptions. As this is a replicated case study, the complete details of the original case study design can be found elsewhere [25]. We provide here concise descriptions of these bugs:

1. Bug #5138[4], described as *"Double-click-drag to select multiple words doesn't work"*.
2. Bug #31779[5], described as *"UnifiedTree should ensure file/folder exists"*.
3. Bug #74149[6], described as *"The search words after ' " ' will be ignored"*.

We identify a feature relevant to each bug description:

- "select multiple words via double-click and drag" for bug #5138;
- "add files and folders to UnifiedTree" for bug #31779
- "search the text in help documentation" for bug #74149.

We refer to these features with a shortened form later in the paper to improve readability: "select", "add files", and "search", respectively.

Table 5 describes the scenarios used to obtain the marked-traces for each feature and the queries formulated (used and formulated in the original experiment [25]).

While executing every scenario, the marked-traces for each scenario contained the following number of unique methods:

- 721 methods for the "select" feature;
- 740 methods for "add files" feature; and
- 771 methods for the "search" feature.

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

**Table 5. Scenarios and queries for locating the features in Eclipse**

| Feature | Simplified scenarios that exercise the features | LSI Query |
|---|---|---|
| **Select** | The programmer opens Java code, ***starts tracing***, double clicks on some Java code and holds the left mouse button, moves the mouse, and releases the mouse button, waits until the first clicked Java word is highlighted, and then ***stops tracing***. | *mouse double click up down drag release*<br>*select text offset document position* |
| **Add files** | The programmer starts Eclipse and creates a file using the file system in a project. Because every other refreshing, the file just created shows or disappears, the programmer traces Eclipse in two scenarios: scenario 1: after refreshing, the file does not show; 2. after refreshing, the file shows. Only traces in the second scenario are collected.<br>In every scenario, the programmer starts Eclipse and creates a file using the file system in a project. He ***starts tracing***, right clicks in the navigator view, the clicks the menu item "refresh", waits until the focus on file disappears (in scenario 1) or shows (in scenario 2), and then he ***stops tracing***. | *unified tree node file system folder location* |
| **Search** | The programmer invokes the help system of Eclipse, he searches with arbitrary words to warm up the system, then he formulates a query which contains unclosed double quote mark, ***starts tracing***, waits until "Nothing found" window appears, and then he ***stops tracing***. | *search query quoted token* |

The first relevant methods identified with SITIR for every feature associated with Eclipse bugs are:

- JavaStringDoubleClickSelector.doubleClicked for the "select" feature;
- UnifiedTree.createChildNodeFromFileSystem for the "add files" feature; and
- QueryBuilder.tokenizeUserQuery for the "search" feature.

Table 6 presents the effectiveness measures for LSI, PROMESIR, SPR, and SITIR. We can observe that the results produced with SITIR are comparable to those obtained with PROMESIR. In each case, the best ranked relevant method was in the second position. Also, the output of the tracing tool used in SITIR, based on marked-traces, is of the same order of magnitude as SPR (note that $E_{SPR}$ in Table 6 is based on the average case scenario, thus, in order to be compared with the output of the SITIR tracing tool, $E_{SPR}$ the numbers should be doubled). We also observe that SITIR significantly outperforms approaches based on SPR or LSI used alone.

**Table 6. Effectiveness of each technique for the Eclipse features**

| Feature | $E_{LSI}$ | $E_{PROMESIR}$ | $E_{SITIR}$ | $E_{SPR}$ |
|---|---|---|---|---|
| **Select** | 7 | 1 | 2 | 268 |
| **Add files** | 2 | 1 | 2 | 170 |
| **Search** | 5 | 3 | 2 | 456 |

## 3.6 Discussion of the Results

As expected, the results of the second case study confirm that SITIR outperforms LSI and SPR in locating bug related features in Eclipse. The SITIR results for the Eclipse case study are very close to the PROMESIR results. The major differences between SITIR and PROMESIR lie in the way tracing is done and how the results of the two types of analyses are combined. Compared to PROMESIR, SITIR requires a single scenario in most cases and only one execution trace to be collected. Note that the size of the set of executed methods in SITIR is comparable with the one obtained with SPR alone, however in order to obtain that, SPR requires multiple (at least two) scenarios. In addition, the combination of the analysis results is more transparent in the SITIR case. The JEdit case study showed that SITIR is significantly less sensitive to poor user queries than LSI alone. We can also see that using marked-traces, not only reduces the size of the traces, but also improves the effectiveness of SITIR. Yet, even with full traces, SITIR gives good results when user queries are refined.

## 3.7 Threats to Validity

In this section we discuss some of the issues that might have affected the results of the case studies and may limit the interpretations and generalizations of the results.

The first issue is the extent to which the software systems used in the case studies are representative of those used in practice. While Eclipse is a real-world program, JEdit is rather average sized. This threat can be reduced if we experimented with other software systems of different sizes taken from other domains.

Another issue is the selection of scenarios to obtain the execution traces using SITIR technique. Since we are not experts in Eclipse and JEdit we can not claim that our scenarios are the best ones to capture the features which are being located. Thus, depending on the chosen scenarios, the results may differ.

The queries formulated to produce the LSI-based rankings are dependent on the programmer's knowledge, thus the final results are also sensitive to user query to some extent.

In our case studies the effectiveness measures for SPR is defined on an average case scenarios (see [25] for details). In reality, the developer may find one of the related methods in the execution trace much faster, for example using search techniques. Since we have the large difference between the SITIR and SPR accuracies,

modifying the formula for computing the effectiveness of these techniques will not drastically change the results.

Finally, the features or bug fixes may be implemented by more methods than those which are suggested in official bug fixes (as in the case of Eclipse). This observation does not impact the results of this case study, since considering more candidate methods will only increase the possibility of identifying one of those methods earlier in the process of feature location.

## 4. Related Work

While existing techniques for feature location broadly fall into three categories based on the type of analysis they use (that is, static, dynamic, and hybrid), we focus here on dynamic and hybrid techniques. A good overview of static techniques is presented in [23].

Wilde and Scully [36] introduced software Reconnaissance based on analyzing overlapping execution traces of test cases, further formalized by Deprez and Lakhotia [8]. In order to identify a feature of interest, the developer needs to formulate at least two test cases, where the first one exercises the desired feature and the second one does not. In its simplest form, the analysis takes the set of software components executed in tests with the feature and subtracts the set of such components executed in the remaining tests. The result contains elements of the source code relevant to the feature of interest. The technique is further developed in [33] and adapted to be applied in distributed systems in [9].

Wong et al. [38] use metrics-based approach to quantify the disparity between a feature and a component, the concentration of a feature in a component, and the dedication of a component to a feature. An extended version of this approach characterizes the distance between features using both a static method and a dynamic one, which takes into account a system operational profile [37].

The Reconnaissance approach is also extended by Antoniol and Guéhéneuc (i.e., SPR) [3] with statistical hypothesis testing based on the events which occur in the marked traces, knowledge-based filtering, and support for multi-threaded applications using processor emulation techniques such as Valgrind for trace collection in C/C++ and Jikes RVM for Java programs.

Eisenberg and De Volder [12] addressed the problem of multiple vs. single traces, by using a complete set of test cases in the target system with one test case per feature. These test cases are partitioned manually into feature-specific subsets which are subsequently used to obtain execution traces. Based on the collected traces for feature-specific test cases, the methods are ranked using heuristic-based criteria.

Eisenbarth et al. [11] proposed a first hybrid technique, by combining static and dynamic analysis to identify features in the source code. The dynamic analysis is performed similarly to Reconnaissance. Formal concept analysis is applied on the resulting execution traces to link the features together, which is used to guide the static analysis. Feature location is performed by means of set operations on concepts, which requires running at least several test cases to identify a single feature.

One of the most recent hybrid approaches [25], PROMESIR combines two existing techniques for feature identification: **S**cenario-based **P**robabilistic **R**anking [3] of events and an information-retrieval-based technique that uses Latent Semantic Indexing [24]. The developer, using SPR, formulates at least two scenarios: one exercising the feature of interest and one not. With the resulting execution traces, SPR produces a set of ranked methods relevant to the feature. In addition, the developer

**Table 7. Summary of feature location approaches, which use dynamic analysis. The techniques that have names are identified with their acronym and a reference, while the other approaches are identified by the first author of the publication where it was presented and the relevant reference. In the case of multiple papers on the same technique we referenced the latest.**

| | DFT [12] | Simmons [33] | SPR [3] | PROMESIR [25] | Edwards [9] | Reconnaissance [36] | Eisenbarth [11] | Exec. slices [38] | SITIR |
|---|---|---|---|---|---|---|---|---|---|
| **Scenarios/traces used per feature** | one | two | two or more | two or more | two or more | two or more | multiple | two or more | one |
| **Trace filtering** | heuristics based on multiple traces | trace intersection | probabilistic ranking of the events + static analysis | probabilistic ranking of the events + IR | weighted relevance of events + causality analysis | trace intersection | FCA + static analysis | trace intersection | IR |
| **Comparison with other approaches** | Recon. | × | grep and FCA | SPR and LSI | × | × | × | × | PROMESIR, SPR, LSI |
| **Tracing technique** | AspectJ instrum. | instrum. | processor emulation | processor emulation and Jikes RVM | instrum. | instrum. | compiled with profiling options – instrum. | instrum. | JPDA |
| **Additional requirements** | large suite of test cases | source code is available | source code is available | source code is available | source code is available | source code is available | source code is available | source code is available | source code is available |
| **Language** | Java | C | C/C++/Java | C/C++/Java | C/C++ | C | C | C | Java |
| **Case study: software systems** | HTMLUnit, HTTPUnit, Axion | Apache | Firefox, Mozilla, Chimera, ICEBrowser, JHtoDraw, XFIG | Mozilla, Eclipse | Gunner, Joint STARS | XREF | XFIG | SHARPE | Eclipse, JEdit |

formulates a query which describes in natural language terms the feature of interest. Using LSI, all the methods in the system are ranked with respect to this query. The rankings of the two approaches, that is, SPR and LSI, are combined via an affine transformation. The authors evaluated this approach on several case studies and the results show that the proposed approach significantly improves the effectiveness of feature location as compared to SPR and LSI techniques if used standalone.

The PROMESIR approach is of interest here as the results in [25] show that dynamic and textual data capture complementary information relevant to the implementation of features. PROMESIR is also the closest technique to SITIR, the main differences consisting in *how many* and *what type* of scenarios are executed, and how the trace data and the LSI-based rankings are *combined*. Table 7 summarizes the main features of all these techniques discussed above, highlighting the features that set them apart from each other.

A number of dynamic approaches exist, which use single traces per feature. They are different from the previous approaches as they focus on identifying multiple features at a time or relationships among them. In particular, these approaches focus on feature interactions [10, 30, 31], feature evolution [15], hidden dependencies among features [13] as well as identifying canonical set of features for a given software system [17].

# 5. CONCLUSIONS AND FUTURE WORK

The results of using SITIR for feature location underscores previous findings, which showed that hybrid techniques for feature location are very effective, especially when applied to large systems. SITIR is unique among the hybrid feature location techniques as it generates a set of methods relevant to a feature of interest, extracting them from a single (marked-) trace. Defining the scenarios is straightforward as they do not have to be very precise. Tracing is unobtrusive for Java programs, with very little execution time overhead. Users can formulate queries that describe in natural language the feature of interest, to rank these methods. Results show that in most cases the relevant methods to the located features rank in the top ten. SITIR is less sensitive to poor user queries as the search using LSI alone. These results are comparable with the state of the art in hybrid techniques for feature location and obtained with less user effort (that is, one trace vs. two or more). All these attributes of SITIR point to a high usability, as programmers in industry may be able to employ these techniques.

Combinations of various complementary techniques for feature location are poised to revolutionize software evolution and open new research directions. We are experimenting with other possible combination of information sources to support feature location and other software evolution tasks. Specifically, we are combining dependency analysis with IR to improve static concept location. The end goal is to devise the best technique to combine all available sources of information used in feature location: execution traces, program dependencies, and textual information.

# 6. ACKNOWLEDGEMENTS

This research was partially supported by grants from the US National Science Foundation (CCF-0438970), US National Institute for Health (NHGRI 1R01HG003491), and the 2006 IBM Eclipse Innovation Awards.

# 7. REFERENCES

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

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, vol. 28, no. 10, October 2002, pp. 970 - 983.

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

[4] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in Proc. of 15th IEEE/ACM International Conference on Software Engineering (ICSE'94), 1994, pp. 482-498.

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

[6] De Lucia, A., Fasano, F., Oliveto, R., and Tortora, G., "Recovering Traceability Links in Software Artefact Management Systems", *ACM Transactions on Software Engineering and Methodology,* 2007.

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

[8] Deprez, J.-C. and Lakhotia, A., "A formalism to automate mapping from program features to code", in Proc. of 8th IEEE International Workshop on Program Comprehension (IWPC'00), 2000, pp. 69-78.

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

[10] Egyed, A., Binder, G., and Grunbacher, P., "STRADA: A Tool for Scenario-Based Feature-to-Code Trace Detection and Analysis", in Proc. of IEEE/ACM 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 41-42.

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

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

[13] Fischer, M., Pinzger, M., and Gall, H., "Analyzing and Relating Bug Report Data for Feature Tracking." in Proc. of IEEE Working Conference on Reverse Engineering (WCRE'03), 2003, pp. 90-101.

[14] Frakes, W. and Kang, K., "Software Reuse Research: Status and Future", *IEEE Transactions on Software Engineering*, vol. 31, no. 7, 2005, pp. 529-536.

[15] Greevy, O., Ducasse, S., and Girba, T., "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", in Proc. of 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 347-356.

[16] Hayes, J. H., Dekhtyar, A., and Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study

of methods", *IEEE Transactions on Software Engineering*, vol. 32, no. 1, January 2006 2006, pp. 4-19.
[17] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., "On Computing the Canonical Features of Software Systems", in 13th IEEE Working Conference on Reverse Engineering (WCRE'06), Benevento, Italy, 2006.
[18] Kuhn, A., Ducasse, S., and Gîrba, T., "Semantic Clustering: Identifying Topics in Source Code", *Information and Software Technology*, vol. 49, no. 3, March 2006, pp. 230-243.
[19] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, 1991, pp. 800-813.
[20] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proc. of 23rd International Conference on Software Engineering (ICSE'01), 2001, pp. 103-112.
[21] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proc. of Automated Software Engineering (ASE'01), 2001, pp. 107-114.
[22] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 4, October 2005, pp. 811-836.
[23] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in Proc. of 13th IEEE International Workshop on Program Comprehension (IWPC'05), 2005, pp. 33-42.
[24] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), 2004, pp. 214-223.
[25] Poshyvanyk, D., Guéhéneuc, G. Y., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June 2007, pp. 420-432.
[26] Rajlich, V., "Changing the Paradigm of Software Engineering", in *Communications of ACM*, vol. August, 2006, pp. 67-70.
[27] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", in *IEEE Software*, 2004, pp. 2-9.
[28] Reiss, S. P. and Renieris, M., "Generating Java Trace Data", in Proc. of the ACM Conference on Java Grande, 2000, pp. 71-77.
[29] Robillard, M., "Automatic Generation of Suggestions for Program Investigation", in Proc. of Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2005, pp. 11 - 20
[30] Salah, M. and Mancoridis, S., "A hierarchy of dynamic software views: from object-interactions to feature-interactions", in Proc. of 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 72-81.
[31] Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M., "Scenario-driven dynamic analysis for comprehending large software systems", in Proc. of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), 2006.
[32] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proc. of International Conference on Aspect Oriented Software Development (AOSD'07), 2007, pp. 212-224.
[33] Simmons, S., Edwards, D., Wilde, N., Homan, J., and Groble, M., "Industrial tools for the feature location problem: an exploratory study", *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 6, 2006, pp. 457-474.
[34] Szegedi, A. and Gyimothy, T., "Dynamic Slicing of Java Bytecode Programs", in Proc. of 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), 2005, pp. 35-44.
[35] Tonella, P. and Ceccato, M., "Aspect Mining through the Formal Concept Analysis of Execution Traces", in Proc. of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), 2004, pp. 112 - 121
[36] Wilde, N. and Scully, M., "Software Reconnaissance: Mapping Program Features to Code", *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.
[37] Wong, W. E. and Gokhale, S., "Static and dynamic distance metrics for feature-based code analysis", *Journal of Systems and Software*, vol. 74, no. 3, February 2005, pp. 283-295.
[38] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S., "Locating program features using execution slices", in Proc. of IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99), 1999, pp. 194-203.
[39] Ye, Y. and Fischer, G., "Supporting Reuse by Delivering Task-Relevant and Personalized Information", in Proc. of IEEE/ACM International Conference on Software Engineering (ICSE'02), 2002, pp. 513-523.
[40] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., "SNIAFL: Towards a Static Non-interactive Approach to Feature Location", *ACM Transactions on Software Engineering and Methodologies*, vol. 15, no. 2, 2006, pp. 195-226.
[41] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June 2005, pp. 429-445.