Feature Location in Source Code: A Taxonomy and Survey

Bogdan Dit, Meghan Revelle, Malcom Gethers, Denys Poshyvanyk
The College of William and Mary

Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system. Many feature location techniques have been introduced that automate some or all of this process, and a comprehensive overview of this large body of work would be beneficial to researchers and practitioners. This paper presents a systematic literature survey of feature location techniques. Eighty-nine articles from 25 venues have been reviewed and classified within the taxonomy in order to organize and structure existing work in the field of feature location. The paper also discusses open issues and defines future directions in the field of feature location.

Keywords: Feature location, concept location, program comprehension, software maintenance and evolution

1. INTRODUCTION

In software systems, a feature represents a functionality that is defined by requirements and accessible to developers and users. Software maintenance and evolution involves adding new features to programs, improving existing functionalities, and removing bugs, which is analogous to removing unwanted functionalities. Identifying an initial location in the source code that corresponds to a specific functionality is known as *feature* (or *concept*) *location* [Biggerstaff'94, Rajlich'02]. It is one of the most frequent maintenance activities undertaken by developers because it is a part of the incremental change process [Rajlich'04]. During the incremental change process, programmers use feature location to find where in the code the first change to complete a task needs to be made. The full extent of the change is then handled by impact analysis, which starts with the source code identified by feature location and finds all the code affected by the change. Methodologically, the two activities of feature location and impact analysis are different and are treated separately in the literature and in this survey.

Feature location is one of the most important and common activities performed by programmers during software maintenance and evolution. No maintenance activity can be completed without first locating the code that is relevant to the task at hand, making feature location essential to software maintenance since it is performed in the context of incremental change. For example, Alice is a new developer on a software project, and her manager has given her the task of fixing a bug that has been recently reported. Since Alice is new to this project, she is unfamiliar with the large code base of the software system and does not know where to begin. Lacking sufficient documentation on the system and the ability to ask the code's original authors for help, the only option Alice sees is to manually search for the code relevant to her task.

Alice's situation is one faced by many software developers needing to understand and modify an unfamiliar codebase. However, a manual search of a large amount of source code, even with the help of tools such as pattern matchers or an integrated development

environment, can be frustrating and time-consuming. Recognizing this problem, software engineering researchers have developed a number of **f**eature **l**ocation **t**echniques (FLTs) to come to aid programmers in Alice's position. The various techniques that have been introduced are all unique in terms of their input requirements, how they locate a feature's implementation, and how they present their results. Thus, even the task of choosing a suitable feature location technique can be challenging.

The existence of such a large body of feature location research calls for a comprehensive overview. Since there currently is no broad summary of the field of feature location, this paper provides a systematic survey and operational taxonomy of this pertinent research area. To the best of our knowledge, Wilde et al. [Wilde'03] is the only other survey, which in contrast to our survey, compares only a few feature location techniques. Our survey includes research articles that introduce new feature location approaches; case, industrial, and user studies; and tools that can be used in support of feature location. The articles are characterized within a taxonomy that has nine dimensions, and each dimension has a set of attributes associated with it. The dimensions and attributes of the taxonomy capture key facets of typical feature location techniques and can be useful to both software engineering researchers and practitioners [Marcus'05b]. Researchers can use this survey to identify what has been done in the area of feature location and what needs to be done; that is, they can use it to find related work as well as opportunities for future research. Practitioners can use this overview to determine which feature location approach is most suited to their needs.

This survey encompasses 89 articles (60 research articles and 29 tool and case study papers) from 25 venues published between November 1992 and February 2011. These research articles were selected because they either state feature/concept location as their goal or present a technique that is essentially equivalent to feature location. The tool papers include tools developed specifically for feature location as well as program exploration tools that support feature location. The case study articles include industrial and user studies as well as studies that compare existing approaches.

There are several research areas that are closely related to feature location, such as traceability link recovery, impact analysis, and aspect mining. Traceability link recovery seeks to connect different types of software artifacts (*e.g.*, documentation with source code), while feature location is more concerned with identifying source code associated with functionalities, not specific sections of a document. Impact analysis is the step in the incremental change process performed after feature location with the purpose of expanding on feature location's results, especially after a change is made to the source code. Feature location focuses on finding the starting point for that change. The main goal of aspect mining is to identify cross-cutting concerns and determine the source code that should be refactored into aspects, meaning the aspects themselves are not known a priori. By contrast, in the contexts in which feature location is used, the high-level descriptions of features are already known and only the code that implements them is unknown. Therefore, articles and research from these related fields are not included here as they are beyond the scope of this focused survey.

The work presented in this paper has two main contributions. The first is a systematic survey of feature location techniques, relevant case studies, and tools. The second is the taxonomy derived from those techniques. An online appendix[1] lists all of the surveyed articles classified within the taxonomy. Section 2 presents the systematic review process. Section 3 introduces the dimensions of the taxonomy, and Section 4 provides brief descriptions of the surveyed approaches. Section 5 overviews the feature location tools

[1] http://www.cs.wm.edu/semeru/data/feature-location-survey/ (accessed and verified on 03/01/2011)

and studies, and Section 6 provides an analysis of the taxonomy. Section 7 discusses open issues in feature location and Section 8 concludes.

2. SYSTEMATIC REVIEW PROCESS

In this paper we perform a systematic survey of the feature location literature in order to address the following research questions (RQ):

- RQ_1: What types of analysis are used while performing feature location?
- RQ_2: Has there been a change in types of analysis used to identify features in source code employed by recent feature location techniques?
- RQ_3: Are there any limitations to current strategies for evaluating various feature location techniques?

In order to answer these research questions, we conducted a systematic review of the literature using the following process (see Figure 1):

- ***Search***: the initial set of articles to be considered during the selection process is determined by identifying pertinent journals, conferences and workshops.
- ***Article Selection***: using inclusion and exclusion criteria the initial set of articles is filtered and only relevant articles are considered beyond this step.
- ***Article Characterization***: articles, which meet the selection criteria, are classified according to the set of attributes that capture important characteristics of feature location techniques.
- ***Analysis***: using the resulting taxonomy and systematic classification of the papers, the research questions are answered and useful insights about the state of feature location research and practice are outlined.

2.1. Search

An initial subset of papers of interest was obtained by manually evaluating articles that appear in different venues considered during our preliminary exploration. We select venues where feature location research is within their respective scope. Also, choosing such venues ensures that selected articles meet some standard (e.g., the papers went through a rigorous peer review process).

2.2. Article Selection

To adhere to the properties of systematic reviews [Kitchenham'04] we define the following inclusion and exclusion criteria. In order to be included in the survey, a paper must introduce, evaluate, and/or complement the implementation of a source code based feature location technique. This includes papers that introduce novel feature location techniques, evaluate various existing feature location techniques, or present tools implementing existing or new approaches to feature location. The papers, which focused on improving the performance of underlying analysis techniques (e.g., dynamic analysis, Information Retrieval), as opposed to the feature location process were excluded.

2.3. Article Classification

The authors read and categorized each article according to the taxonomy and the attributes presented in Section 3. The process of classifying the articles was followed by four authors individually. Using initial classifications produced by the authors we identified papers that had some disagreements and further discussed those papers. The set of attributes was extracted and defined by two of the authors. Having all four authors characterize the articles allows us to verifying the quality of the taxonomy, minimizing potential bias. In certain cases disagreements served as an indication that our taxonomy and attributes or their corresponding descriptions required refinement. Through this

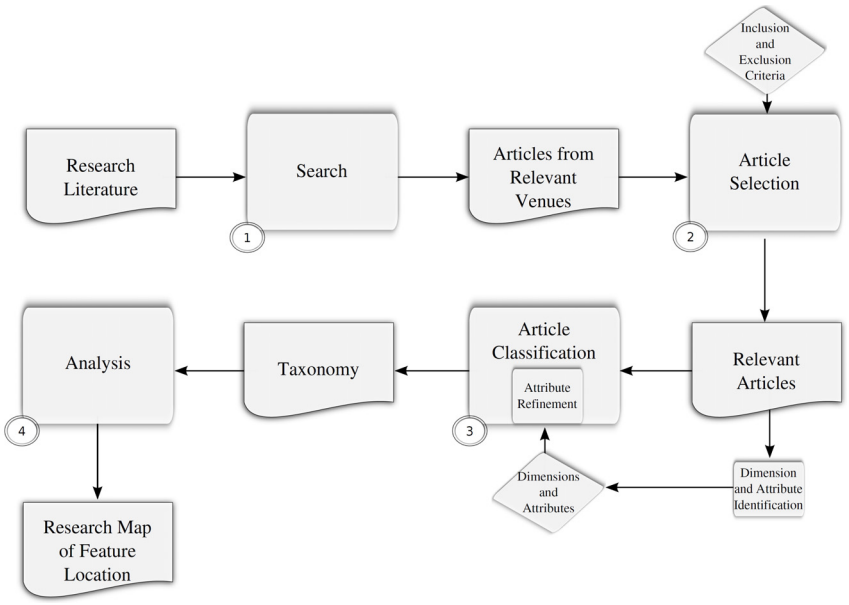

Figure 1 Systematic review process

process we were able to improve the quality of our taxonomy and attribute set as well as improve their descriptions.

2.4. Analysis

Following the process of classifying research papers our final step includes analysis the results, answers to the research questions as well as an outline of future directions for researchers and practitioners investigating feature location techniques. In order to complete this step we analyzed the trends in our resulting taxonomy and observed interesting co-occurrences of various attributes across feature location techniques. We also investigated characteristics that rarely apply to the set of techniques considered as well as characteristics which are currently emerging in the research literature.

3. DIMENSIONS OF THE SURVEY

The goal of this survey is to provide researchers and practitioners with a structured overview of existing research in the area of feature location. From a methodical inspection of the research literature we extracted a number of key dimensions[2]. These dimensions objectively describe different techniques and offer structure to the surveyed literature. The dimensions are as follows:

- *The type of analysis*: What underlying analyses are used to support feature location?
- *The type of user input*: What does a developer have to provide as an input to the feature location technique?
- *Data sources:* What derivative artifacts have to be provided as an input for the feature location technique?

[2] Some of these dimensions were discussed at the working session on Information Retrieval Approaches in Software Evolution at 22nd IEEE International Conference on Software Maintenance (ICSM'06): http://www.cs.wayne.edu/~amarcus/icsm2006/

- *Output*: What type of the results and how are they provided back to the user?
- *Programming language support*: On which programming languages was this technique instantiated?
- *The evaluation of the approach*: How was this feature location technique evaluated?
- *Systems evaluated*: What are the systems that were used in the evaluation?

The order in which these dimensions are presented does not imply any explicit priority or importance.

Each dimension has a number of distinct attributes associated with it. For a given dimension, a feature location technique may be associated with multiple attributes. These dimensions and their attributes were derived by examining an initial set of articles of interest. They were then refined and generalized to succinctly characterize the properties that make feature location techniques unique, and can be used to evaluate and compare them. The goal of the taxonomy's dimensions and attributes it to allow researchers and practitioners to easily locate the feature location techniques that are most suited to their needs. The dimensions and their associated attributes that are used in the taxonomy of the surveyed articles are listed in Table 1. These dimensions and attributes are discussed in the remainder of this section. The *attributes* are highlighted in italics.

3.1. Type of Analysis

A main distinguishing factor of feature location techniques is the type, or types of analyses they employ to identify the code that pertains to a feature. The most common types of analyses include dynamic, static, and textual. While these are not the only types of analysis possible, they are the ones utilized by the vast majority of feature location techniques, and some approaches even leverage more than one of these types of analysis. In Section 4, descriptions of all the surveyed articles are given, and the section is organized by the type(s) of analysis used.

Dynamic analysis refers to examining a software system's execution, and it is often used for feature location when features can be invoked and observed during runtime. Feature location using dynamic analysis generally relies on a post-mortem analysis of an execution trace. Typically, one or more feature-specific scenarios are developed that invoke only the desired feature. Then, the scenarios are run and execution traces are collected, recording information about the code that was invoked. These traces are captured either by instrumenting the system or through profiling. Once the traces are obtained, feature location can be performed in several ways. The traces can be compared to other traces in which the feature was not invoked to find code only invoked in the feature-specific traces [Eisenbarth'03, Wilde'95]. Alternatively, the frequency of execution portions of code can be analyzed to locate a feature's implementation [Antoniol'06, Eisenberg'05, Safyallah'06]. Using dynamic analysis for feature location is a popular choice since most features can be mapped to execution scenarios. However, there are some limitations associated with dynamic analysis. The collection of traces can impose considerable overhead on a system's execution. Additionally, the scenarios used to collect traces may not invoke all of the code that is relevant to the feature, meaning that some of the feature's implementation may not be located. Conversely, it may be difficult to formulate a scenario that invokes only the desired feature, causing irrelevant code to be executed. Dynamic feature location techniques are discussed in Section 4.2.

Static analysis examines structural information such as control or data flow dependencies. In manual feature location, developers may follow program dependencies in a section of code they deem to be relevant in order to find additional useful code, and

this idea is used in some approaches to feature location [Chen'00]. Other techniques analyze the topology of the structural information to point programmers to potentially relevant code [Robillard'08]. While using static analysis for feature location is very close to what a developer searching for code may do, it often overestimates what is pertinent to a feature and is prone to returning many false positive results. Static approaches to feature location are summarized in Section 4.3.

Textual approaches to feature location analyze the words used in source code. The idea is that identifiers and comments encode domain knowledge, and a feature may be implemented using a similar set of words throughout a software system, making it possible to find a feature's relevant code textually. Textual analysis is performed using three main techniques: pattern matching, Information Retrieval (IR) and natural language processing (NLP). Pattern matching usually involves a textual search of source code using a utility, such as grep[3]. Information Retrieval techniques, such as Latent Semantic Indexing (LSI) [Deerwester'90], Latent Dirichlet Allocation (LDA) [Blei'03] and Vector Space Model (VSM) [Salton'86], are statistical methods used to find a feature's relevant code by analyzing and retrieving identifiers and comments that are similar to a query provided by a user. A good overview of applications of IR techniques in Software Development, Maintenance and Evolution can be found in [Binkley'10a, b]. NLP approaches can also exploit a query, but they analyze the parts of speech of the words used in source code. Pattern matching is relatively robust, but not very precise because of the vocabulary problem [Furnas'87]; the chances of a programmer choosing query terms that matches the vocabulary of unfamiliar source code are relatively low. On the other hand, NLP is more precise than pattern matching but much more expensive. Information Retrieval lies between the two. No matter the type of textual analysis used, the quality of feature location is heavily tied to the quality of the source code naming conventions and/or the user-issued query. Textual feature location techniques are reviewed in Section 4.4.

Feature location is not limited to just dynamic, static, or textual analysis. Many techniques draw on multiple analysis methods to find a feature's implementation, and some do not use any of these types of analyses. Existing approaches that combine two or more types of analysis do so with the goal of using one type of analysis to compensate for the limitations of another, thus achieving better results than standalone techniques. The unique ways in which multiple types of analysis are combined for feature location are described in Sections 4.5 through 4.8. Other approaches do not rely on dynamic, static, or textual analysis. For instance, two feature location techniques rely on historical analysis to mine version control systems in order to identify lines of code [Chen'01a] or artifacts related to a feature [Cubranic'05]. Another technique examines the code visible to a programmer during a maintenance task and tries to infer what was important [Robillard'03a]. These alternative approaches are explained in Section 4.9.

3.2. Types of User Input

A FLT ultimately assists developers during software maintenance tasks. This dimension describes the type of input that a developer has to provide for the FLT. The input can be a *query*, *an execution scenario*, a *source code artifact*, or a combination of these. The query can be a set of words that describe a bug or a feature that the developer is attempting to locate. The query can be generated in several ways. It can be compiled by the developer, it can be suggested by the FLT supporting automatic query expansion, or it can be extracted from the description of the feature report or some other

[3] http://www.gnu.org/software/grep/ (accessed and verified on 03/01/2011)

documentation artifacts. The execution scenario is a set of steps, which the developer has to perform on an instrumented software system in order to exercise a feature of interest,

Table 1 Dimensions and attributes of the feature location taxonomy. The attributes are highlighted in bold, and their description is given after the colon separator

Dimension	Attribute: Description
Type of analysis	**Dynamic:** Dynamic analysis is used to locate features **Static:** Static analysis is used to locate features **Textual:** Textual analysis is used to locate features **Historical:** Information from software repositories is used to locate features **Other:** Another type of analysis is used to locate features
User input	**Natural Language Query:** A textual description of a feature (or a bug report description) or a user specified query is used as an input **Execution Scenario:** The developer uses a scenario in order to exercise a feature (or reproduce a bug) of interest in order to collect execution traces **Source Code Artifact:** A software artifact is used as a starting point (or seed) for the feature location technique
Data sources (derivative inputs)	Source code: • **Compilable/executable:** An executable program is used as an input to extract derivative information • **Non-compilable:** A source code that may or may not contain errors that prevent it to be compliable is used as an input to extract derivative information Derivative analysis data from source code: • **Dependence Graph:** The dependence graph can be derived directly from the source code (i.e., static) or it can be derived from execution information (i.e., dynamic) • **Execution Trace:** A set of methods that were executed when exercising a scenario • **Historical Information:** Historical information from source code repositories about the changes in the source code. This information includes log messages and actual changes • **Other:** Another source of information is used for feature location
Output	Source code: • **File/class:** The results produced by the FLT are at file/class level granularity • **Method/function:** The results produced by the FLT are at method/function level granularity • **Statement:** The results produced by the FLT are at statement level granularity **Non-source code artifact:** The output produced by the FLT is a non-source code artifact – e.g., bug report Presentation of the results: • **Ranked:** The results produced by the feature location technique have scores that can be used to rank the results based on their relevance to the user input • **Visualization:** The results produced by the feature location technique are presented using an advanced way of visualizing information
Programming language support	**Java:** The approach supports feature location in software written primarily in Java **C/C++:** The technique can find features for software systems written in C/C++ **Other:** Feature location in some other language is supported, e.g., Fortran, Cobol
Evaluation	**Preliminary:** The evaluation is on small systems or small datasets or only preliminary evidence is given (i.e., proof of concept) **Benchmark:** The evaluation uses a dataset that was published by other authors or the dataset used in this evaluation is later used by other researchers Human subjects: • **Academic:** Students or non-professional developers participated in evaluation • **Professional:** Professional developers participation in evaluation of the results **Quantitative:** The results were evaluated via comparative metrics such as precision, recall, etc. **Qualitative:** The paper discusses details about the characteristics of the technique/tool and/or some aspects of the results **Comparison with other approaches:** Comparisons of the author's approach with existing solutions **Unknown/none:** There is no evaluation performed, or the details are not available
Systems evaluated	The software systems upon which the FLT has been applied are listed

with the purpose of collecting execution information about the system (i.e., execution trace). The source code artifact can be any artifact, such as a class or method, which the developer chooses as a starting point for the feature location technique. The rationale is that the FLT will perform some analysis starting from that artifact (e.g., a method in source code) and it will return other artifacts (e.g., source code methods) related to it. As mentioned before, based on the FLT, the developer has to provide as an input any of these types of inputs, or a combination of these.

3.3. Data Sources (Derivative Inputs from Software)

In addition to the type of user input, the FLT might require other sources of information such as the source code or derivative data from source code. Some techniques require the source code to be compilable or executable, in order to extract static dependencies or execution information, whereas other techniques that use textual information could use as input source code that contains errors and therefore may not necessarily be compilable. On the other hand, some techniques require artifacts derived from source code, such as dependence graphs, execution traces, and historical information from source code repositories concerning source code changes (i.e., change log messages and concrete source code changes). We also include the attribute *other* for the information extracted from bug repositories, forums, documentation, test cases, etc. In addition, the attribute *other* can denote the fact that the technique requires some feedback from the developer. Typically, the sources of utilized information reflect the type of analysis that can be employed. Dynamic analysis uses execution traces captured when a feature is executed. Different representations of source code, such as a program dependence graph, can be used by static analysis FLTs. Source code and documentation can be leveraged in textual analysis to find words that are relevant to a feature. Analysis applied on change history data uses version control systems, issue trackers, communication archives, etc., in order to leverage past changes in software, as a way of supporting feature location.

3.4. Output

Once a FLT identifies candidate software artifacts for a given feature, those results must be presented to the developer. The results can have different granularity levels and different presentations.

The types of granularities are *classes/files*, *methods or functions*, and *statements* (i.e., basic blocks, lines of code, variables, etc.). Throughout this survey, we refer to portions of source code at any level of granularity as **program elements**. The more fine-grained the program elements located by a technique, the more specific and expressive the feature location technique is. For instance, when the results are presented at the statement level granularity all the basic blocks or variables may be relevant, but when results are presented at class level granularity, not all the methods from the class may pertain to the feature. Some approaches may be applicable to multiple levels of granularity, but only those program elements that are actually shown to be supported in an article are reported in this survey. On the other hand, some FLTs may produce artifacts that are not part of the source code, such as bugs, documentation, etc. We categorize these results in the *non-source code artifacts* attribute.

In this survey we also distinguish among different ways the results can presented to software developers. For example, one option is to present a list of candidate program elements ranked by their relevance to the feature [Antoniol'06, Eisenberg'05, Liu'07, Marcus'04, Robillard'08]. We use the attribute *ranked* to denote such a presentation of the results. Another way in which feature location results are presented is as an unordered set of program elements [Eaddy'08a, Eisenbarth'03, Wilde'95]. In other words, a set of

elements is identified as being relevant to a feature, but no notion of their degree of relevance is provided. If the *ranked* attribute is not specified, we assume that a FLT presents the results in this way. Another form of presenting the results is by using *visualization* to highlight relevant program elements [Bohnet'07b, Walkinshaw'07, Xie'06]. Note that the ranked and visualization attributes are not mutually exclusive. Finally, some feature location techniques do not automatically identify relevant program elements but describe a process that a programmer can follow to manually search for a feature's implementation [Chen'00]. Since different feature location techniques present their results in different ways, comparing approaches that use dissimilar reporting styles can be challenging.

3.5. Programming Language Support

The programming language in which a software system is written can play a factor in the types of feature location techniques that can be applied to it. Textual and historical analyses are programming language agnostic at the file level, but require parsers if applied at fine granularity levels (e.g., method level granularity). Static and dynamic analyses may be limited due to tool support for a given platform or a programming language. In this survey, all programming languages on which a technique has been applied are reported. The majority of existing feature location approaches have been exercised on Java or C/C++ systems since ample tool support is available for these languages. Other programming languages that have been supported include FORTRAN and COBOL. Knowing the languages under which an approach works can help researchers and practitioners select an appropriate technique, though the fact that an approach has not been used on a certain programming language does not imply that it is not applicable to systems implemented in that language.

3.6. Evaluation

The way in which a feature location technique is evaluated provides researchers and practitioners with useful information on the approach's quality, effectiveness, robustness, and practical applicability. Evaluating a feature location technique is difficult because defining the program elements that are relevant to a feature may be subjective at times. Despite this difficulty, researchers have devised a number of approaches to assess feature location techniques. Evaluations of traditional software engineering techniques are classified as *survey*, *experiment*, *case study* and *none* [Wohlin'99]. However, this standard categorization does not apply to our taxonomy for two reasons. First, due to the fact that the feature location field is not as matured as other software engineering fields, there are no papers that fall into the categories survey and experiment. In addition, most of the papers evaluate their approaches using case studies or they only have a basic (i.e., preliminary) evaluation, which means that according to the standard categories, all the papers would be categories as either case studies or none. Our second reason is that we wanted to categorize the papers in our survey using finer grained attributes, which would give more insights into the evaluation used in the papers. Thus, we choose the attributes *preliminary*, *benchmark*, *human subjects*, *quantitative*, *qualitative*, *comparison with other approaches* and *unknown/none*, which are described next. Once again, by using these attributes we wanted to distinguish as much as possible between different evaluations in different papers.

The most simplistic evaluations are *preliminary* in nature and involve small systems or a few data points, and their purpose is to provide an anecdotal evidence that the approach works (i.e., proof of concept).

More advanced evaluations use *benchmarks* that contain detailed information and datasets that could be used by other researchers in their evaluation. These datasets could contain a list of features, textual description or documentation about the features, mappings between features or bugs and program elements that are relevant to fixing the bug or implementing the feature (referred to as *gold sets* in the literature), patches submitted to an issue tracker, etc. Benchmarks carry more weight than anecdotal evaluation, because they can be used by other researchers in other approaches and the results could be compared in a fair way. In other words, if two approaches are applied on the same dataset, their results could be easily compared to establish which technique produces more accurate results, for instance. However, if the two techniques are evaluated on different datasets (even if these datasets are part of the same software system), the comparison of the results is biased by the difference in those datasets. In this survey we categorize papers as using benchmarks if their evaluation is using the same datasets or a subset of these datasets, which were made available by other authors. We also categorize the evaluation of papers that initially used that dataset and made it available as benchmarks.

One of the difficulties in using benchmarks in the evaluations is that very few of them are made available up to date. One of the reasons is that constructing benchmarks requires substantial efforts and the results are not guaranteed to be one hundred percent correct and complete. For example, if benchmarks are constructed from source code repositories or from patches extracted from issue repositories, there is no assurance that they contain all the information required to implement the feature or fix the bug (unless this information is verified by the original designers and developers of the software system). They might contain noise, or they may only pertain to a small portion of the feature and not touch all of its program elements. Another way to evaluate a feature location approach is to have system experts or even non-experts assess the results, which is an evaluation method often used by IR-based search engines. When multiple experts or non-experts are used, the intersection of their subjective evaluations can be used to create a benchmark. However, the agreement among programmers as to what program elements are relevant to a feature has been shown to be low in some cases [Robillard'07b].

In this survey we also distinguish between evaluations that use human subjects. These developers could have an *academic* background, such as undergraduate or graduate students, or they could be *professional* developers that come from industry. Other ways to differentiate the evaluations is based on their *qualitative* and/or *quantitative* results. In a quantitative evaluation, the technique is evaluated in terms of some metrics, such as precision, recall, effectiveness, etc., whereas a qualitative evaluation provides details about some particular aspect of the technique or the results. Papers which do not contain any evaluation, or for which the details are not known are marked as *unknown/none*. It is important to note that these evaluation attributes are not mutually exclusive. In other words, one approach could have both a qualitative and a quantitative evaluation, whereas others could have only one of them. Or for example, one evaluation could be preliminary, but it could contain some qualitative information. One of the most important attributes of this dimension is *comparison with other approaches*. Ideally, when new feature location techniques are introduced, they should be directly compared with existing approaches in order to demonstrate their (expected) superior performance. Articles that include comparisons of feature location techniques are very useful to researchers and practitioners because they highlight the advantages and limitations of the compared approaches in certain settings. Feature location techniques that appear frequently in comparisons are Abstract System Dependence Graphs (ASDG) [Chen'00], Dynamic

Feature Traces (DFT) [Eisenberg'05], Formal Concept Analysis-based feature location (FCA) [Eisenbarth'03], LSI-based feature location [Lukins'08, Marcus'04], Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) [Poshyvanyk'07a], software reconnaissance [Wilde'95], and Scenario-based Probabilistic Ranking (SPR) [Antoniol'06]. UNIX grep is also another popular point of comparison because programmers often use it to search for relevant code.

3.7. Software Systems used for Evaluation

A wide variety of software systems have been studied in feature location research, and the size and type of systems used in a case study reflect, to a degree, the applicability of a technique. By reviewing the software systems that have previously been used for feature location, some patterns emerge. Some of the more popular systems are web browsers like Mozilla[4], Firefox[5], Mosaic, and Chimera[6]. Other systems that have been investigated frequently are Eclipse[7], jEdit[8], and JHotDraw[9]. For some of these systems, there are a few datasets that are repeatedly used, but the majority of papers evaluate their techniques on datasets extracted from these systems, and which are never used in other evaluations. In other words, there is no de facto benchmark proposed and we argue that there should be one. Beside these popular systems, an abundance of other software systems have been studied. The systems on which a feature location technique has been applied are listed in the taxonomy. Having a comprehensive list of the software systems studied for feature location allows researchers to identify good candidates for systems to use in their own evaluations, or even better, to build benchmarks for these systems and make them publicly available. In addition, it allows practitioners to recognize approaches that may be successfully applied to their own software if the program they wish to apply a feature location technique to is similar to a system on which the approach has already been used.

3.8. Other Attributes not Included in the Taxonomy

While classifying the papers based on this taxonomy, we initially considered including the dimensions, such as *tool availability* and *reproducibility*, each with their two binary attributes *yes* and *no*. For example, if the paper introduces a new tool that is made publicly available for others to use, we would mark that attribute accordingly. Similarly, if the evaluation provided enough details about the design of the study so that other researchers could reproduce it, we would mark the evaluation of the paper as reproducible. However, it turned out that actual categorization was quite subjective, and there was a considerable amount of disagreement between the authors of the survey, that we decided to exclude these dimensions. In order to categorize the papers based on these dimensions a panel of experts should evaluate these artifacts (e.g., steps towards this model have been already done for accepted research papers at ESEC/FSE 2011[10] by Artifact Evaluation Committee).

4. SURVEY OF FEATURE LOCATION TECHNIQUES

This section describes in detail the systematic process used to survey the literature as well as summarizes the 89 articles reviewed for this survey.

[4] http://www.mozilla.org/ (accessed and verified on 03/01/2011)
[5] http://www.mozilla.org/firefox (accessed and verified on 03/01/2011)
[6] http://www.chimera.org/ (accessed and verified on 03/01/2011)
[7] http://www.eclipse.org/ (accessed and verified on 03/01/2011)
[8] http://www.jedit.org/ (accessed and verified on 03/01/2011)
[9] http://www.jhotdraw.org/ (accessed and verified on 03/01/2011)
[10] http://2011.esec-fse.org/cfp-research-papers (accessed and verified on 03/08/2011)

Table 2 Venues which have published the articles included in this survey

Type	Acronym	Description
Journal	JSME	Journal on Software Maintenance and Evolution: Research and Practice
	JSS	Journal on Systems and Software
	TOSEM	ACM Transactions on Software Engineering and Methodology
	TSE	IEEE Transactions on Software Engineering
Conference	AOSD	Aspect-Oriented Software Development
	APSEC	Asia Pacific Software Engineering Conference
	ASE	International Conference on Automated Software Engineering
	CSMR	European Conference on Software Maintenance and Reengineering
	ESEC/FSE	European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering
	ICSE	International Conference on Software Engineering
	ICSM	International Conference on Software Maintenance
	IWPC/ICPC	International Workshop/Conference on Program Comprehension
	VISSOFT	International Workshop on Visualizing Software for Understanding and Analysis
	WCRE	Working Conference on Reverse Engineering

4.1. Performing Systematic Review Process

In this section we provide details on how we performed the systematic review process to institute our survey. We start this process with an initial *search* phase which includes the identification of highly relevant venues. More specifically, we selected journals, conferences, and workshops papers where research on feature location is within their respective scopes. Table 2 lists the abbreviations and names of the venues.

After performing our preliminary search for relevant venues, in the *article selection* phase, we manually identified literature which meets our inclusion criteria while filtering articles using our exclusion criteria (see Section 2.2). Titles, abstracts, keywords and entire articles were cautiously inspected to determine whether or not they should be included in the survey. The two authors of this paper were responsible for identifying suitable articles using the selection criteria, however, the selections were later confirmed during the article classification phase by all the authors.

Figure 2 shows the distribution of articles across the venues. The height of the bars represents the number of feature location articles published. Venues at which only one surveyed paper was published are grouped together in the "Other" bar. Filled bars represent journals, and gray bars denote conferences and workshops.

Using the taxonomy presented in the previous section the selected articles were classified. During the *article classification* phase all four authors separately classified each of the selected research papers. For each dimension, the authors identified whether or not the attributes of the taxonomy were applicable to the research papers. Attribute selection for a given dimension is not exclusive. That is, multiple attributes may apply to a given dimension while classifying a particular paper. Following the categorization of the papers by all the authors an agreement was computed. Initially there were a few disagreements between author's classifications, however, following a meeting in which

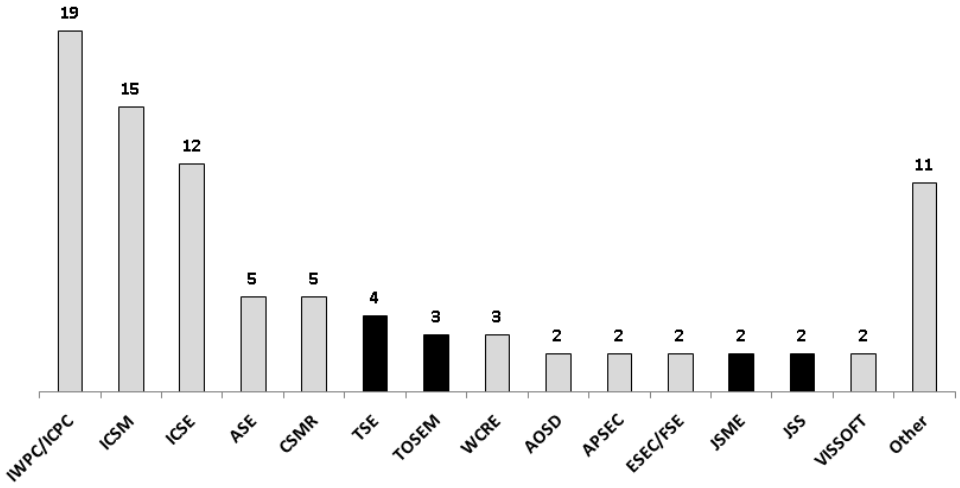

Figure 2 Distribution of the surveyed articles. Black bars represent journals and gray bars denote conferences. The values above the bars represent the number of feature location papers published in that venue

authors discussed their reasoning we were able to reach a consensus on the classification of all the papers. Additionally, the discussion helped revising some details of the taxonomy and description of the attributes by identifying ambiguous descriptions.

The remainder of Section 4 is devoted to summarizing the feature location techniques using the taxonomy as a means of structuring the discussion. More specifically, the dimension *type of analysis* is used to organize the remaining subsections. When summarizing the feature location techniques and multiple articles (i.e., conference and journal versions) describe a given approach and share identical classifications, both are cited but the summary primarily pertains to the latest version. The articles are classified by the types of analysis used for feature location, and other dimensions of the taxonomy are mentioned whenever applicable. The type of analysis/analyzes employed is the most distinguishing characteristic of feature location approaches, so it is a logical choice for decomposing the survey. In the subsections below, the surveyed articles are categorized by their use of one or more types of analysis: dynamic; static; textual; dynamic and static; dynamic and textual; static and textual; dynamic, static, and textual; and other. The discussion for each paper includes a brief characterization of the approach, distinguishing features from other approaches, tool support, and some details of the evaluation. Table 3, Table 4 and Table 5 (located after the references) present the articles and their classification within the dimensions of the taxonomy. Table 3 and Table 4 present the approaches, whereas Table 5 presents the tools and the case studies.

4.2. Dynamic Feature Location

Dynamic feature location relies on collecting information from a system during runtime. Dynamic analysis has a rich research history in the area of program comprehension [Cornelissen'10], and feature location is one subfield in which it is used. A number of dynamic approaches exist that deal with feature interactions [Egyed'07, Salah'04, Salah'06], feature evolution [Greevy'05, '06], hidden dependencies among features [Fischer'03], as well as identifying a canonical set of features for a given software system [Kothari'06]. These techniques are beyond the scope of this survey which focuses only

on approaches that seek to identify candidate program elements that implement a feature. This subsection summarizes articles that achieve this goal using dynamic analysis.

Software reconnaissance [Wilde'92, Wilde'95] is one of the earliest feature location techniques, and it relies solely on dynamic information. Two sets of scenarios or test cases are defined, such that some scenarios activate the feature and the other scenarios do not, and then execution traces of all the scenarios are collected. For example in a word processor, if the feature to be located is spell checking, feature-specific scenarios would activate the spell checker and the other scenarios would not. Feature location is then performed by analyzing the two sets of traces and identifying the program elements (methods) that only appear in the traces that invoked the feature, using both Deterministic and Probabilistic Formulations. This idea of comparing traces from scenarios that do and do not invoke a feature has been heavily used and extended by other researchers in the field. The software reconnaissance approach has been implemented in tools such as RECON2 and RECON3[11]. This approach was evaluated on small C systems, using a small number of scenarios, and the results show that the software reconnaissance is useful in indicating small portions of code where developers should look at. A study involving professional developers [Wilde'95] showed evidence that software reconnaissance could be easily adopted by professional developers. In terms of potential limitations of this technique, the results produced are heavily influenced by the quality of the scenario and traces provided. In addition, approach can be used in finding a subset of the artifacts that correspond to a particular feature, but it cannot guarantee in finding all of the related artifacts for a feature. In other words, it provides a very good starting point. Finally, this technique cannot be applied for finding features that are always present in the program, and for which there are no test cases that can and cannot exercise that feature.

An extension of the software reconnaissance approach is Dynamic Feature Traces (DFT) [Eisenberg'05], which works as follows. The developer provides several scenarios or test cases that exercise the features in order to collect execution traces. Next, all pairs of method callers and callees are extracted from the traces, and each method is assigned a rank for the feature. The rank is based on the average of three heuristics: multiplicity, specialization, and depth. Multiplicity is the percentage of a feature's tests that exercise a method compared to the percentage of methods in each non-feature's set of tests. Specialization is the degree to which a method was only executed by a feature and no others. Depth measures how directly a set of tests exhibits a feature compared to the other test sets. This approach was implemented in a prototype tool. This paper also presents a preliminary evaluation on three Java systems where DFT and the software reconnaissance technique were compared. The results of this study revealed that software reconnaissance is not as effective as DFT in finding relevant code. Since this technique is an extension of software reconnaissance, its limitations are similar.

Wong et al. [Wong'99] proposed a technique based on execution slices, which is able to produce results at a finer level of granularity, such as statements, basic blocks, decisions or variable uses, as opposed to producing results at method level-granularity. This technique requires as input a few test cases that exercise and a few test cases that do not exercise the feature of interest. The dynamic information extracted from the instrumented system, on which the test cases were run, consists of the statements that were executed, as opposed to a list of methods. Using this information, this technique is able to distinguish between code that is unique to a feature and code that is common to several features. The tool that implements this approach is called χVue [Agrawal'98], and it supports program instrumentation and collecting execution information. In addition it provides a visualization of the statements relevant to a feature. The evaluation of this

[11] http://www.cs.uwf.edu/~recon/ (accessed and verified on 03/01/2011)

technique based on execution slices was performed on five feature of a small C system called SHARPE [Sahner'86]. The results of the evaluation indicate that the output produce by the technique can be used by developers as a starting point for analyzing the feature of interest. In addition, based on a qualitative analysis of the results, the authors hypothesize that using static information might enhance the results.

Eisenbarth et al. [Eisenbarth'01b, Eisenbarth'01c] introduced a technique for generating feature component maps utilizing dynamic information. Given execution traces resulting from various usage scenarios (covering a set of features of a system) concept analysis is applied to reveal relationships between features and components in addition to feature to feature relationships. Based on the resulting concept lattice a feature component map is derived and used to identify components of interest given a particular feature. Additionally, a case study is performed on using Xfig as the subject software system demonstrating the value of their approach for generating feature component maps.

Safyallah and Sartipi [Safyallah'06] introduced an approach that applies a data mining technique on the execution traces. This technique consists of analyzing using sequential pattern mining a set of execution traces, which are collected from a set of feature specific scenarios. This technique is able to identify continuous fragments of execution traces, called execution patterns, which appear in at least a given number (i.e., *MinSupport*) among all the execution traces. These execution patterns undergo a set of refinements based on adjusting the *MinSupport* threshold. The results of this approach are a set of continuous fragments of execution traces which correspond to a particular feature. A preliminary case study was performed on a medium size C system, called Xfig[12], and the results show that this approach is able to identify a set of core methods that are specific to the input feature. This technique has the advantage of reducing the complexity of analyzing large traces, by identifying their relevant parts. In addition, the sequence of operations from the trace is not altered, and this approach is even able to locate execution patterns specific for less visible features, such as mouse pointer handling, canvas view updating, and other. An extension of their approach [Sartipi'10] allows the extracted execution patterns to be distributed over a concept lattice, in order to distinguish the common group of functions from the feature-specific group of functions. The advantage of using the concept lattice is that it provides developers the chance of identifying a family of closely related features in the source code. An evaluation of this new approach on two C programs, Xfig and Pine[13], showed promising results for identifying feature specific functions, as well as common functions.

A feature location technique designed specifically for distributed systems is introduced by Edwards et al. [Edwards'06]. This technique aims at reducing the imprecision that tends to be associated with existing dynamic FLTs applied on multithreaded or distributed systems. The imprecision of existing techniques stems from the stochastic nature of distributed systems, and from the inability of the technique to identify correctly the order and the time events happen. To overcome this problem, Edwards et al. proposes a definition of time intervals based on causal relationships among events (messages). For example, events are order temporally in a single process, and the events from different processes are causally ordered by message passing. The technique not only requires execution information about the system as input, but it also requires the developer to identify the first and last event associated with a feature. The technique identifies all the events that causally follow or precede a feature's starting and ending events, respectively. This sequence of events is called an interval. The output produced

[12] http://www.xfig.org/ (accessed and verified on 03/01/2011)
[13] http://www.washington.edu/pine/(accessed and verified on 03/01/2011)

by the technique is a ranked list of program elements which are assigned a component relevance index, which is the proportion of executions of that element which occur during a feature's interval. A preliminary evaluation on a large scale distributed system indicate that the technique is able to distinguish between feature related code and not feature related code, based on the component relevance index associated with each program element. The technique is useful for any distributed system for which the program instrumentation does not significantly alter their behavior.

Bohnet et al. [Bohnet'08b] proposed a technique that allows developers to visualize various characteristics of the execution information in order to gain insight into how features are implemented in the code. The developer provides as input a scenario that triggers a feature of interest in the system in order to collect an execution trace. The execution trace is used as an input for an analysis tool and various characteristics of the execution trace are presented to the developer using advanced visualization views. These views are synchronized among each other, which mean they allow to simultaneously present the same information extracted from the trace from different perspectives. For example, the portions of the trace that are highlighted by the developer and which have a correspondence in the source code are also highlighted. A change in a view updates the information in the other views. The approach was compared against grep in a preliminary evaluation involving a large C/C++ software system. In this evaluation, grep was shown to be unpractical, because it returned too many results, whereas using the tool a developer was able to locate the concept of interest in less than half an hour and without having prior knowledge about the system. The advantage of this technique is that the advanced visualization allows the developer to manage the complexity associated with a large execution traces.

One of the main shortcomings of FLTs based on dynamic analysis is in the overhead it imposes on a system's execution. In distributed and time-sensitive systems, the use of dynamic analysis can be prohibitive. Edwards et al. [Edwards'09] report on their experiences using dynamic analysis to perform feature location in time-sensitive systems. Instrumenting a software system in order to collect execution traces of the program elements that are invoked affects the system's runtime performance. Edwards et al. developed a minimally intrusive instrumentation technique called *minist* that reduced the number of instrumentation points while still keeping test code coverage high. For an initial evaluation, Apache's httpd[14] and several large in-house programs were used, and *minist* was compared to uninstrumented executions as well as several other tools for collecting traces. The *minist* approach increased execution time by only 1% on httpd, while the other tracing tools caused increases of 7% to over 2,000%.

Another systematic survey of feature location approaches utilizing execution information can be found in a comprehensive survey of approaches to program comprehension via dynamic analysis [Cornelissen'10].

4.3. Static Feature Location

In contrast to dynamic feature location, static feature location does not require execution information about a software system. Instead, its source code is statically analyzed and its dependencies and structure are explored manually or automatically. Some static FLTs leverage several types of control and data dependencies. Other static techniques use the structure of a software system's dependencies. In general, the static FLTs require not only a dependence graph, but also a set of software artifacts which serve as a starting point for the analysis in order to generate program elements relevant to the initial set. The initial set of artifacts is usually specified by the developer.

[14] http://httpd.apache.org/ (accessed and verified on 03/01/2011)

Chen and Rajlich [Chen'00] introduced the concept of Abstract System Dependence Graphs (ASDG), which are based on abstracting the System Dependence Graphs (SDG). The nodes of an ASDG are either functions or global variables, and the edges between nodes represent either control dependencies between functions or data flow between variables. The static concept location technique that uses ASDGs require as input a statically built dependence graph as well as a starting node which is selected by the developer. The starting node can be a program element relevant to the feature, which is a priori known by the developer, or it can be a randomly selected node, or it can be the node corresponding to the main method. This FLT requires at each step feedback from the developer, which investigates a node and decides if it is relevant to the feature or not. The FLT keeps track of the search graph (i.e., visited components and their neighbors) and based on the developer's feedback about the relevant and irrelevant nodes, it updates and expands the search graph and tries to propose only relevant nodes. This process continues until the developer has found all the program elements related to the maintenance task. The tool that supports this static FLT based on ASDGs is called Ripples [Chen'01b]. Ripples is not only able to generate the ASDG from the source code of C programs, but it also visualizes the graphs and allows the developer to input her feedback about which node is relevant. Another tool implementation of this static FLT is called JRipples [Buckner'05]. This tool is an Eclipse plug-in, which supports feature location on Java systems. Unlike Ripples, JRipples does not have visualization, but it provides support for impact analysis and change propagation. The static FLT introduced by Chen and Rajlich [Chen'00] was evaluated on the Mosaic web browser. The results show that among the 984 functions in Mosaic, the developer performing concept location on a maintenance task was able to partially comprehend the system by investigating only 22 (2%) of the functions.

Robillard and Murphy [Robillard'02, '07a] developed the Concern Graphs representation, which allows the representation of a concern (or feature) in an abstract way. This abstraction allows creating and storing mappings between features and source code. A Concern Graph encapsulates a subset of program elements and a set of relations between them. These relations are based on the static dependencies between the program elements. The tool that was implemented to support Concern Graphs is called FEAT (Feature Exploration and Analysis Tool). FEAT allows developers to visualize the Concern Graphs, to explore the Concern Graph and examine the source code associated with the program elements of the graph, and to allow developers to alter the Concern Graph by adding or removing program elements and relations. In an evaluation involving developers that were required to locate a concern in a Java system, the Concern Graphs representation was shown to be appropriate for expressing the concerns during maintenance tasks, as well as for manipulation or analysis of the concerns. In addition, Concern Graphs are easy and intuitive to use even by unfamiliar developers, and they can be used even for industrial-sized systems.

Robillard [Robillard'05a, Robillard'08] introduced an approach that analyses the topology of structural dependencies in a program in order to propose relevant program elements for the developer to investigate. One of the differences between this approach and ASDGs is that it requires less interaction from the developer. Robillard's approach takes as input a set *I* of program elements which are marked as relevant by the developer. The approach examines the structural dependencies of the elements in *I* and the rest of the system and produces a suggestion set *S*. Both the input and output sets are fuzzy sets, which means that a program element is part of that set with only a degree of certainty. In other words each element from the set has associated a value that signifies its relevance. The relevance values are based on two metrics, namely *specificity* and *reinforcement*. A

program element's specificity measure is inversely proportional with the number of program elements related to it. In other words, a program element is more specific than other if it has fewer program elements related to it. A program element's reinforcement measure is directly proportional with the number of elements of interest related to it. A tool that supports tool the management of concerns which are scattered throughout the code is called ConcernMapper[15] [Robillard'05b]. This Eclipse plug-in can be used by programmers to specify the initial set of interest relevant to a task. The tool that actually implements the FLT via topology analysis is called Suade[16] [Weigand-Warr'08]. An evaluation on a few medium-size java systems show that given an initial set of program elements, the approach is able to return a ranked list of program elements, where the top entries are highly relevant to the initial program elements. In other words, the approach is able to suggest program elements worthy of investigation to developers and at the same time it can avoid suggesting less interesting (irrelevant) ones.

Saul et al. [Saul'07] introduced FRAN (Finding with RANdom walks), an approach that recommends a set of related program elements (e.g., methods) given a specific starting point as an input. FRAN only uses the structural information about a system (e.g., method call graph) and conceptually FRAN generalizes Robillard's approach [Robillard'05a]. This is because FRAN also takes as an input a program element *e* that has some interest to the developer, and builds a program dependence graph of the neighborhood of *e*. The main difference is that FRAN uses a larger set of related program elements and ranks them using the scores produced by applying a random walk algorithm [Kleinberg'99] to the program dependence graph (as opposed to ranking the elements using the specificity and reinforcement metrics). An evaluation on the Apache HTTPD system was performed and FRAN was compared against Suade and FRIAR (Frequent Itemset Automated Recommender), an approach inspired from Association Rule Mining which ranks program elements based on the support values. The results showed that FRAN produced better results (in terms of returning relevant methods) than FRIAR and Suade.

While ASDGs and topology analysis use both control and data dependencies to some extent, Trifu [Trifu'08] introduced an approach to feature location based only on static dataflow analysis. The input for this approach is a set of variables, called information sinks, that are selected by the developer and which are used as starting points to identify all the parts of the source code where the values from the variables propagate. This is done by tracking the dataflow dependencies in the code. Because the granularity of the input program elements is more fine grained (i.e., variables), the results are also more fine grained than other FLTs. The tool that implements this approach is called CoDEx (Concern Discrimination and Explorer), which allows developers to mark the variables of interests and presents the parts of the source code which are related to the input variables. An initial evaluation was performed on JHotDraw. For this evaluation, the information sinks selected were all the variables with no outgoing dataflow paths, and the approach grouped the 6,049 variables into 310 concerns (features). A manual inspection of a few of these concerns revealed that program elements grouped under the same concern are highly relevant to that concern (feature). Trifu [Trifu'09] improved this dataflow based concern identification approach by introducing the concept of information sources, which define boundaries for a concern.

[15] http://www.cs.mcgill.ca/~martin/cm/ (accessed and verified on 03/01/2011)
[16] http://www.cs.mcgill.ca/~swevo/suade/ (accessed and verified on 03/01/2011)

4.4. Textual Feature Location

Source code comments and identifiers are embedded within textual information about features of the systems. Feature location techniques based on textual analysis are aiming at establish a mapping between the textual description of a feature given by the developer and the parts of the source code where that feature is implemented. The approaches to establish the mapping between the description of the feature and the source code include textual search with *grep* [Petrenko'08], Information Retrieval [Cleary'09, Gay'09, Marcus'04, Poshyvanyk'07b], and natural language processing [Hill'09, Shepherd'07].

One simple way of searching for code that is relevant to a task is by using a textual search that describes that task. For example, developers formulate a query that describes the feature they are looking for and then use a tool such as grep to find and investigate lines of code that match the query. Petrenko et al. [Petrenko'08] developed a feature location technique based on *grep* and ontology fragments. The ontology fragments store partial domain knowledge about a feature. The hypothesis of this approach is that ontology fragments help developers formulate queries and guide their investigation of the results, which would increase the effectiveness of the feature location. As programmers gain more knowledge of the system, the ontology fragments can be refined and expanded. The tool used to support the management of the ontology fragments is called Protégé[17]. An exploratory study on two large systems, Eclipse and Mozilla, showed that for locating a bug in the code, ontology fragments required, on average, a few source code methods in Mozilla and Eclipse to be inspected. These results were comparable to other feature location techniques [Liu'07, Poshyvanyk'06a] in which programmers also only had to examine about ten methods.

Wilson [Wilson'10] extended Petrenko et al.'s approach, by introducing a systematic approach for formulating queries based on ontology fragments, which represent partial knowledge about the system. Using an ontology fragment, the developer can formulate a query based on the terms that are present in the ontology fragment, and it can provide that query as an input for *grep*. A preliminary evaluation involving four developers that were required to perform concept location on the Mozilla and Eclipse systems reveal the fact that only a small and partial knowledge about the system is sufficient for successfully locating a concept in the code.

Information Retrieval (IR) is a more advanced technique that can be used instead of the traditional *grep* pattern matching. Marcus et al. [Marcus'04] use Latent Semantic Indexing [Deerwester'90] to map the feature descriptions expressed in natural language by developers to source code. LSI is an advanced IR technique that infers relations between words and passages in large bodies of text. A corpus is created from extracting all identifiers and comments from the source code of a system. The corpus is preprocessed by splitting compound identifiers based on common naming conventions. The corpus is partitioned into documents representing all terms associated with a program element. Documents can be of different granularities, such as classes or methods. The corpus is then transformed into an LSI subspace through Singular Value Decomposition (SVD). After SVD, each document in the corpus has a corresponding vector. To search for code relevant to a feature, a programmer formulates a query consisting of terms which describe the feature. The query is also transformed into a vector, and a similarity measure between the query vector and all the document vectors is used to rank documents by their relevance to the query. The similarity measure is known as the cosine similarity because it computes the cosine between the query and document vectors. The

[17] http://protege.stanford.edu/ (accessed and verified on 03/01/2011)

output produced by this FLT is a list of methods ranked by their textual similarity with the developer input query. The approach was evaluated on the Mosaic web browser, using the same feature used in the study by Chen and Rajlich [Chen'01b]. The new approach was compared against grep and ASDGs, and several advantages were found. LSI is as easy to use as grep, yet it produces better results. Also, LSI was able to identify some relevant program elements missed by ASDGs. Recently, LDA has been applied for bug localization [Lukins'08, Lukins'10]. The proposed approach was compared to LSI and has been shown to be an effective alternative to using LSI for concept location.

Poshyvanyk and Marcus [Poshyvanyk'07b] added Formal Concept Analysis (FCA) to the feature location that uses LSI. FCA takes as input a matrix specifying objects and their associated attributes and then produces clusters, called concepts, of the objects based on their shared attributes. These concepts can be organized hierarchically in a concept lattice. In this case, the objects are methods and the attributes are words that appear in the source code of those methods. To combine the two types of analyses, LSI's ranked results are clustered using FCA. The top *k* attributes of the first *n* methods ranked by LSI are used to construct FCA's input matrix and create a lattice. Nodes in the lattice have associated attributes (terms) and objects (methods), and programmers can focus on the nodes with attributes similar to their query to find feature-relevant methods. The new feature location technique based on FCA was compared against the FLT that uses LSI alone on two maintenance tasks of Eclipse, and the results show that the new approach is able to group relevant information using concept lattices, which means that developers can locate a concept in code by analyzing fewer methods, as opposed to the case where the results are presented as a ranked list.

Cleary and Exton [Cleary'07, Cleary'09] also use IR for feature location, but their solution incorporates non-source code artifacts, such as bug reports, mailing lists, external documentation, etc. Their approach, called cognitive assignment, considers indirect correspondences between query and document terms so that relevant source code can be retrieved even if it does not contain any of the query terms. Queries are expanded by analyzing term relationships from both source code and non-source code artifacts. This approach was implemented by extending the cognitive assignment Eclipse plug-in [Cleary'06] to incorporate the expansion queries mechanism. A case study was conducted on Eclipse in which cognitive assignment was compared to other IR techniques, such as language modeling [Zhai'04], dependency language model [Gao'04], vector space model [Salton'86], and LSI. The results show that cognitive assignment matches the performance of the other IR techniques and in some cases it outperforms them.

The results of any textual feature location technique are heavily influenced by the quality of the queries used. In other words, a textual FLT could produce more accurate results if the developer formulates more accurate queries, by refining or modifying existing ones. Gay et al. [Gay'09] introduce the notion of relevance feedback into textual feature location with IR. Relevance feedback incorporates user input to improve IR results. After IR returns a ranked list of program elements relevant to a query, the developer rates the top *n* results as relevant or irrelevant. Then a new query which incorporates the developer feedback is automatically formulated and new results are returned, and the process repeats. A case study was performed in which a single developer was asked to use IR and relevance feedback to locate the source code associated with change requests (representing features) in Eclipse, jEdit, and Adempiere[18]. Each change request had associated with it a patch that was used to implement the change request, and these patches were used to generate the gold set of

[18] http://sourceforge.net/projects/adempiere/ (accessed and verified on 03/01/2011)

methods used in the evaluation. The results indicate that relevance feedback is more effective and efficient than a pure IR-based approach.

Similarly to Information Retrieval, Independent Component Analysis (ICA) [Comon'94] can examine source code text to identify features and their implementations [Grant'08]. ICA is a signal analysis technique that separates a set of input signals into statistically independent components. To apply ICA for feature location, a term by document matrix is constructed in which the rows correspond to methods, columns represent terms, and cells contain the frequency of a term in a method. ICA factors the matrix into two new matrices. The first new matrix, called the source signal matrix, stores independent signals which can be thought of as features. The second new matrix, the mixing matrix, holds information about how relevant each signal is to a method. Unlike feature location with LSI, feature location with ICA does not need a query for a specific feature since it seeks to identify multiple independent signals (features) at once. A preliminary evaluation on a medium size C program showed that ICA was able to identify a few key concepts from the code.

Textual feature location techniques are not limited to using IR only. Shepherd et al. [Shepherd'06] proposed a technique that leverages information about the use of verbs and their direct objects (nouns) in source code identifiers to create a natural language representation of the code called an Action-Oriented Identifier Graph (AOIG). In the AOIG, all the verb-direct object (verb-DO) pairs are extracted from the code and a mapping between each verb-DO and the code is kept. An Eclipse plug-in called ViRMoVis was created to implement this approach. The developer formulates a query in the form of a verb, and ViRMoVis suggests a set of direct objects associated with that verb. The developer then selects the appropriate direct objects (i.e., refines the query), and the tool displays all the uses of the selected verb-DO pairs. An exploratory study on the Java JHotDraw system revealed that using ViRMoVis a developer was able to locate a feature by examining only a small number of classes and methods.

Hill et al. [Hill'09] also used NLP and the idea of query expansion and refinement in their approach to feature location based on contextual searching. Instead of focusing on verbs and direct objects, their analysis centers on three types of phrases: noun phrases, verb phrases, and prepositional phrases. The phrases are extracted from method and field names and additional phrases are generated by also looking at a method's parameters. Once the phrases are extracted, they are grouped into a hierarchy based on partial phrase matching. The phrases are linked to the source code from which they were extracted. An Eclipse plug-in and a PHP script were created to implement this approach. A user looking for a particular feature formulates a query and the tool searches the extracted phrases for matches. The result returned to the user is a hierarchy of phrases and the method signatures associated with them, giving some context to the results. This approach was evaluated on Rhino, on a subset of features from the benchmark created by Eaddy et al. [Eaddy'08b], as well as on a subset of features used in the evaluation by Shepherd et al. [Shepherd'07]. For the evaluation, 22 developers (17 of them with 1 to 9 years of industry experience) assessed the two approaches, and the results show that contextual search has been shown to significantly outperform the verb-direct object approach both in terms of effort (number of queries needed) and effectiveness (f-measure).

Abebe and Tonella [Abebe'10] introduced an approach that extract concepts from source code by applying NLP techniques. The identifiers from the program elements are extracted and candidate sentences that use those identifiers are formed. Some of the sentences that do not follow certain rules are eliminated, and the remaining sentences are used as an input for creating ontologies that capture the concepts and the relations from the source code. The identifiers represent the concepts, and the semantics of the sentences

establish the relations in the ontology. A preliminary evaluation on the WinMerge system revealed that enhancing the queries using information from the ontologies increases the precision of concept location, by allowing developers to formulate more precise queries and by reducing the search space. Approach by Abebe and Tonella's is relevant to Petrenko et al.'s [Petrenko'08] approach, but the main difference is that the former approach automatically generates the ontologies, whereas for the latter approach the ontologies were generated manually by developers.

Würsch et al. [Würsch'10] leveraged static analysis and semantic web-based technologies to provide users with a natural language guided query interface to answering program comprehension questions. More specifically, Resource Description Framework (RDF) is used to model source code entities and their properties/relationships resulting in RDF graphs. The Web Ontology Language (OWL) is used to model an ontology for source code. This allows developers to query source code with some natural language guided vocabulary within their IDE. Developers can form questions such as "What method calls ...?" and "What attributes have the type ...?". The authors conducted a case study with JFreeChart to demonstrate the usefulness of their technique. With that, the authors provided some first evidence that their technique is capable of answering questions in natural language form.

4.5. Combined Dynamic and Static Feature Location

The combination of dynamic and static analysis is a well-known and powerful combination in other areas of research such as testing and program analysis [Dufour'07, Ernst'03]. This combination has also been applied for feature location. Dynamic analysis can be used to reduce the search space to only those program elements that were executed in a trace, and then static analysis can work on the smaller set of program elements to rank them or find additional relevant elements.

Eisenbarth et al. [Eisenbarth'01a, '03] proposed an approach that clusters the execution information collected about the system using concept analysis. Several execution traces that exercise different features are collected from the instrumented system. The information from the traces is extracted and used as an input for concept analysis, which treats the methods as objects and the features invoked during the execution scenario as attributes. The result of concept analysis is a concept lattice which can be investigated by the developer in order to identify candidate program elements that are solely relevant to a feature or contribute to a feature but are also used by other features. The program elements located by this approach are only a subset related to a feature (i.e., a starting point), and developers seeking additional relevant code can follow an approach similar to ASDGs. In a preliminary evaluation on two C browsers, Mosaic and Chimera, the approach was able to recover a partial description of the software architecture responsible to implement the set of features used as input. The results also show that out of the large number of methods from the browsers, very few methods needed to be inspected manually. Koschke and Quante [Koschke'05] adapted this approach to collect input traces at statement level granularity, which means the approach is able to locate features at the level of basic blocks (as opposed to method-level granularity). An evaluation conducted on two compilers, sdcc and cc1, confirmed the findings of the previous approach, and proved that this FLT could be used in practice. The advantage of combining concept analysis with dynamic information is that the concept lattice will handle the differences among the traces that arise during the invocation process.

To overcome the imprecision and noise associated with collecting dynamic data, Antoniol and Guéhéneuc [Antoniol'05, Antoniol'06] introduced the Scenario-based

Probabilistic Ranking (SPR) approach, which combines both dynamic and static data for identifying a feature's relevant program elements (methods). The idea behind SPR is to assign for each event from an execution trace a probability of that event being associated with a feature and then rank all the events. In SPR, similar to software reconnaissance, two sets of scenarios are defined, scenarios that do and do not exercise a feature, and method-level execution traces are collected for each scenario. Intervals correspond to a subsequence of contiguous events (method calls) from the traces, where I is an interval from a relevant scenario, and I' is an interval from an irrelevant scenario. Events are classified as relevant to a feature or not by determining if their frequency in interval I is greater than their frequency in interval I'. For any interval, an event's frequency is computed as the ratio of the number of times the event appears in an interval over the total number of events in the interval. Essentially, determining whether an event is relevant to a feature or not is a statistical hypothesis test. The null hypothesis is that an event's frequency in the two types of intervals is the same. A threshold, Θ, is chosen, and if an event is classified as relevant to a feature more than Θ times, the null hypothesis is rejected with a confidence level α. Events are also ranked by their relevance to a feature using a relevance index score that is computed from the number of times an event appears in relevant intervals versus the number of times it appears in irrelevant intervals. The source code is represented as an Abstract Object Language (AOL) using static analysis. This format that represents the program's architecture is used for highlighting the ranks of the elements identified using the dynamic analysis. The elements that appear in the trace that exercised or not the feature of interest will highlight the program architecture differently, which means it will be easy to compare these architecture to find out which program elements are part of the feature. SPR has been applied to a number of systems including Mozilla, Firefox, Chimera, ICEBrowser, JHotDraw, and Xfig. Case studies have compared SPR directly to feature location using grep, and the concept analysis based approach [Eisenbarth'01a, '03]. The results show that because SPR ranks its results, it is successful at reducing the amount of data that a programmer needs to investigate. In addition, SPR allows developers to visualize the micro architectures of the system.

Rohatgi et al. [Rohatgi'07, Rohatgi'08, Rohatgi'09] introduced an approach that locates features in source code at class level granularity. More specifically, their technique takes as input an execution trace and a class or component dependency graph (CDG) for feature location based on impact analysis. Distinct classes are extracted from a feature-specific execution trace, and then the CDG is used to rank the classes by the impact a change to them would have on the software system. The hypothesis of this technique is that classes with the least amount of impact are most likely related to the feature. In a preliminary evaluation on two Java systems, Weka[19], a machine learning tool and Checkstyle[20], a tool for formatting source code, the approach was able to identify and rank appropriately classes relevant to the feature. However, in few cases, the relevant classes were not ranked correctly.

Walkinshaw et al. [Walkinshaw'07] developed a feature location technique based on call graph slicing, which uses the concepts of landmarks and barriers. The first step of the approach is to identify landmark and barrier methods in a static call graph, where a landmark is a method that contributes to a feature and barriers are irrelevant methods. Direct paths between landmark nodes, known as hammock graphs, are found, and additional dependencies are obtained via backward slicing. Barriers and their

[19] http://www.cs.waikato.ac.nz/ml/weka/ (accessed and verified on 03/01/2011)
[20] http://checkstyle.sourceforge.net/ (accessed and verified on 03/01/2011)

dependencies are removed from the call graph to prevent exploration of irrelevant methods. The output of this approach is a pruned call graph. The technique was evaluated on NanoXML[21], Freemind[22], and JHotDraw, finding that the landmark and barrier technique substantially reduces the size of the call graph that a programmer has to investigate.

4.6. Combined Dynamic and Textual Feature Location

Dynamic and textual analyses are very synergistic when it comes to their use in feature location. Dynamic analysis generally yields good recall, while textual analysis has good precision. Their combination may lead to improved results over individual techniques. Both analyses can be used to rank program elements by their relevance to a feature, so a logical next step is to combine both of the rankings produced by these techniques. Another combination of dynamic and textual analyses is to use dynamic analysis to filter the program elements for textual analysis instead of ranking all the program elements in a software system.

Poshyvanyk et al. [Poshyvanyk'06a, '07a] introduced the Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) approach, which performs feature location by combining "expert" opinions from two existing feature location techniques. The first is Scenario based Probabilistic Ranking (SPR) [Antoniol'05, Antoniol'06] and the second is information retrieval with LSI [Marcus'04]. Both approaches rank program elements according to their relevance to the feature of interest. Those rankings are combined through an affine transformation to produce PROMESIR's results. The weight given to SPR or LSI can be varied to reflect the amount of confidence that should be assigned to each of the experts. An evaluation on Eclipse and Mozilla indicate that PROMESIR outperforms the two techniques on which it is based. It is interesting to note that one of the datasets used in the evaluation was exactly the same as the one used by Antoniol et al. in their evaluation of SPR.

Similar to PROMESIR, the Single Trace and Information Retrieval (SITIR) [Liu'07] approach is a feature location technique that applies information retrieval on the execution information collected from exercising a single scenario relevant to the feature. In other words, the technique takes as an input a developer query and a scenario relevant to the feature and produces a list of methods which are ranked based on the similarity with the query. The novelty of this approach is that it ranks only the methods that appear in the execution trace, as opposed to ranking all the methods from the system. This innovation reduces dramatically the search space and yields better results. An Eclipse plug-in that supports this approach, FLAT^3 [Savage'10b], was later developed. An evaluation on jEdit and Eclipse compared SITIR against the IR based FLT, SPR and PROMESIR. The results showed that in general, SITIR ranked the relevant methods higher than the other approaches.

Asadi et al. [Asadi'10] proposed a feature location technique that identifies cohesive and decoupled fragments from execution traces which are related to concepts. The approach takes as input a scenario that exercises a feature of interest in order to collect the execution trace. The trace is preprocessed in order to remove irrelevant methods to the feature (e.g., mouse tracking methods) and to compress the trace (i.e., remove repetition of methods) for an easier analysis. The preprocessed trace is used as an input for a genetic algorithm that separates the trace into fragments that contain methods that are highly cohesive and which are highly decoupled with other fragments. The fitness function used in the genetic algorithm is based on the conceptual cohesion metric defined

[21] http://devkix.com/nanoxml.php (accessed and verified on 03/01/2011)
[22] http://freemind.sourceforge.net/ (accessed and verified on 03/01/2011)

by Marcus and Poshyvanyk [Marcus'05a]. A preliminary evaluation on two Java systems, ArgoUML and JHotDraw showed that the approach was able to locate concepts with high precision. One of the drawbacks of this technique observed in the evaluation was that precision tended to drop for different features that used similar sequences of methods.

Revelle et al. [Revelle'10] proposed a feature location technique that combines textual information with the results produced by applying advanced link analysis algorithms on execution information. The approach takes as an input a query and an execution scenario that exercises a feature. A program dependence graph is generated from the execution trace by using the caller and callees methods as nodes and the relations between them as edges. Two link analysis algorithms (i.e., PageRank [Brin'98] and HITS [Kleinberg'99]) are applied on the program dependence graph, which associate for each node in the graph (i.e., method from trace) a score, based on the nodes relative importance in the graph. Any of the PageRank scores, the HITS Authorities or the HITS Hubs scores could be used to rank the methods from the execution trace. We refer to this novel FLT as WM. An alternative is to filter the methods that are ranked on top or bottom using the link analysis scores from the methods that appear in the execution trace, and after that to rank the remaining methods using the textual similarity between the methods and the developer's query. We refer to this novel FLT as IR_{LSI}WM. The evaluation of this approach was performed on two Java systems, namely Eclipse and Rhino. The Rhino data used in the evaluation represents a subset of features from the benchmark created by Eaddy et al. [Eaddy'08b]. In the evaluation the WM and the IR_{LSI}WM approaches were compared against each other as well as against the LSI based FLT [Marcus'04] and the SITIR [Liu'07] approach. The results showed that WM produced more accurate results than the LSI approach, but did not perform as good as the SITIR approach. On the other hand, the IR_{LSI}WM outperformed all the other approaches.

Hayashi et al. [Hayashi'10a] proposed iFL, an approach that combines static and dynamic analysis, along with relevance feedback to identify source code entities which comprise a feature of interest. The paper claims that the iterative approach leads to improved query formulation by end users and the evaluation of relevance during the iterative process enhances a users understanding of features implemented in a given software system. iFL requires as input source code, a test case (used to derive dynamic dependencies), a query, and hints (relevance feedback) and returns to the user source code entities ordered by their respective evaluation scores. Evaluation of the tool is performed using five change requirements of Sched and two change requirements of JDraw systems. The results, which compare the interactive and non-interactive versions of the approach, indicate that the iterative technique is capable of reducing the understanding cost (based on improvement of lowest ranked relevant method).

4.7. Combined Static and Textual Feature Location

Several researchers have combined static and textual analyses for feature location. This combination is a natural choice because either textual analysis can be used to reduce the overestimation that static analysis is prone to produce or static analysis can be used to find additional candidate program elements given a starting set of highly relevant ones from textual analysis. Thus, combining these two types of analysis has the potential to yield better results than either static or textual analysis alone.

A static, non-interactive approach to feature location (SNIAFL) is introduced by Zhao et al. [Zhao'04, '06]. SNIAFL uses Information Retrieval in conjunction with a branch-reserving call graph (BRCG), essentially an expanded version of a call graph with branch information. An initial set of program elements (i.e., methods) specific to the feature is located using information retrieval, and then additional relevant elements are found using

the BRCG. The initial set is produced by using the vector space model to obtain and rank methods by their similarity to a query. A gap threshold technique is used to find the largest difference between the similarities of consecutive ranked methods. The methods above this gap are considered to be the initial elements specific to the feature. From the initial set, the BRCG is pruned to remove branches that are not in the initial set. Also, the relevance of branches that are included in the initial set is propagated through the graph's dependencies, essentially generating a static pseudo-execution trace. In case studies on two pieces of GNU software, SNIAFL had better precision and recall than both a pure IR approach and a purely dynamic approach, lending evidence to the fact that combining static and textual analyses is more successful than using them as standalone approaches.

Similar to the SNIAFL approach, Dora [Hill'07] combines static and textual analysis to perform feature location. Programmers formulate a query which is used to compute a method relevance score that is based on the term frequency-inverse document frequency of words that appear in the name and body of the method. Then, starting from a seed method selected by the developer, Dora follows static caller/callee edges to identify additional relevant methods using the relevance score. Dora was evaluated on a number of open source Java systems and compared to Suade and two naïve textual and static approaches. The datasets for these systems were created via a user study [Robillard'07b] in which programmers were asked to locate the implementations of several features. Dora was found to be the most successful technique in the evaluation.

In Dora and SNIAFL, one type of analysis is used to prune another. Shao and Smith [Shao'09] combine information retrieval and static control flow information in a different manner for feature location. First, LSI is used to rank all the methods in a software system by their relevance to a query. Then, for each method in the ranked list, a call graph is constructed. A method's call graph is inspected to assign it a call graph score. The call graph score counts the number of a method's direct neighbors that also appears in LSI's ranked list. Finally, the method's cosine similarity from LSI and its call graph score are combined using an affine transformation, and a new ranked list is produced. A preliminary evaluation compared the technique against LSI on a C++ program called iVistaDesktop, which simulates Microsoft's Windows Vista operating system.

Ratiu and Deissenboeck [Ratiu'06, '07] introduced an approach that recovers the mapping between the real world concepts and the relevant parts of the source code. Their approach is not explicitly aimed at feature location but at interpreting programs from the point of view of the domain knowledge they implement, which could be features. They developed a framework that describes semantic defects caused by improper naming and an algorithm to recover the mappings between ontology elements and program elements. The algorithm maps concepts and program elements via graph matching. Concepts are represented in the ontology and programs are abstracted as graphs. The framework and algorithm have been applied to the Java standard library, finding actual examples of semantic defects.

Shepherd et al. [Shepherd'07] employ natural language processing in conjunction with static analysis for feature location. The premise of their approach involves the observation that in source code, actions are represented by verbs, and nouns correspond to objects. Their approach, which is implemented in the tool Find-Concept, has the following steps: initial query formulation, query expansion, and a search of the action-oriented identifier graph model (AOIG). The developer creates a query consisting of a verb and a direct object. Then, Find-Concept expands the query using NLP and its knowledge of the terms used within the software's source code to recommend new queries. Once the user refines the query, the tool locates nodes in the AOIG that contain a verb and direct object from the query and returns the methods to which they are

mapped. Find-Concepts uses program analysis to identify any dependencies between the methods returned by the AOIG search and then presents the user with a visualization of the results as a graph. In a user study, Find-Concept's verb-direct object approach was compared to the lexical search provided by the Eclipse IDE and Google Eclipse Search [Poshyvanyk'06c] on a suite of open-source Java systems and overall was found to be the most effective search technique, without requiring additional effort from the users.

Hayashi et al. [Hayashi'10b] combined textual analysis and static analysis with domain ontologies to link user specified sentences to source code fragments. Their approach returned ordered functional call-graphs (i.e., methods and their invocations of the input source code) extracted for the initial call graph of the software system under investigation. Given the source code of a software system, a sentence, and an ontology (i.e., directed relations between words) the proposed technique obtains a call graph of the system using static analysis, extracts the terms from the source code and the sentence, and subsequently traverses the call graph in search of functional call graphs. Identification of function call graphs involves using terms of the sentence to identify root nodes. Following the identification of root nodes, paths to traverse are determined by locating entities which contain terms from the user provided sentence. The resulting functional call graphs are prioritized according to the importance determined by their name, using relations in the ontology, and the ratio of words in the input sentence. Evaluation of the technique was performed using seven features of JDraw. A comparison of precision and recall indicated that the use of an ontology provided better results than the case where the ontology was not used.

4.8. Combined Dynamic, Static, and Textual Feature Location

Cerberus [Eaddy'08a] is a feature location technique that utilizes three types of analysis: dynamic, static, and textual. Currently, it is the only approach that leverages all three types of analysis. At the core of Cerberus is a technique called *prune dependency analysis* (PDA), whereby a relationship between a program element and a feature exists if the program element should be removed or modified if the feature were to be pruned from the software system. Given an initial set of relevant elements to be pruned, PDA infers additional relevant elements. Cerberus uses PROMESIR to combine rankings of program elements from execution traces with rankings from information retrieval to produce seeds for PDA. Cerberus' authors created a large benchmark for Rhino[23], an open source Java implementation of JavaScript, in which the code for over 400 features defined in the system's documentation were manually located. This benchmark was used to evaluate and compare Cerberus to software reconnaissance, SPR, DFT, LSI, finding that combining the three types of analysis was the most effective approach.

4.9. Other Feature Location Techniques

CVSSearch [Chen'01a] is a feature location technique that uses textual and historical information from CVS repositories. CVS comments generally describe the change made to the lines of code which are being committed, and those comments typically hold true for many future revisions of the software. The tool maps the lines of code that were changed during CVS commits with the CVS comments associated with those commits. This means that if a line of code was changed in multiple commits, it will have associated all the CVS comments from those commits. The tool requires as input a query, and CVSSearch[24] returns all lines of code whose associated comments contain at least one of

[23] http://www.mozilla.org/rhino/ (accessed and verified on 03/01/2011)
[24] http://cvssearch.sourceforge.net/ (accessed and verified on 03/01/2011)

the query words. The textual search is done using grep. Each returned line also has a score indicating how well it matches the developer query. In a user study involving 74 students that were required to performed concept location on a few programs from the KDE suite, CVSSearch was compared against grep. The results of the evaluation showed that even though the CVS comments are a valuable source of information, the CVSSearch tool that exploits them complements the traditional grep technique, but it does not replace it.

Hipikat [Cubranic'03, Cubranic'05] is a feature location tool that also makes use of archival information for feature location, but instead of identifying candidate program elements, Hipikat recommends artifacts from a project's archives such as online documentation, versions, bugs, or communications. Hipikat forms a group memory from a project's history as recorded in source code repositories, issue trackers, communication channels, and web documents [Cubranic'04]. Links between these artifacts are inferred using IR. For example, a source code version can be linked to a bug report if the bug's id is included in a repository commit log message. This history is used to find relevant artifacts in response to a user query. The query consists of an artifact, potentially a program element, for which the user wants recommendations of related artifacts. Hipikat responds with a list of artifacts ranked by their relevance. The tool has been used in two case studies. In the first, Hipikat was validated on AVID[25], and in the second, it was used to aid programmers performing a change task on Eclipse.

Robillard and Murphy [Robillard'03a] propose a unique approach to feature location that automatically analyses a transcript of a program investigation session in an integrated development environment. The transcript records which program elements were visible to a developer during a maintenance task and how they were accessed: through a code browser, following a cross-reference, recalling an open window or tab, scrolling, or keyword search. For each event in the transcript, all visible program elements are determined. Then, for each visible element, a probability that it is the element in which the programmer was interested in is assigned to it. The probabilities are based on weights associated with each event type. Next, a correlation metric is calculated between all pairs of program elements. The correlation is based on how closely two elements were accessed in the transcript. Finally, concerns (features) are generated by clustering program elements, and the concerns can be named and saved for later retrieval.

Similarly to CVSSearch, Ratanotayanon et al. [Ratanotayanon'10] implemented Kayley, a tool that utilized historical and textual information for the task of feature location. Ratanotayanon et al. introduced the concept transitive change-set which is used to enrich the set of source code entities associated with a commit of a version control system. Transitive change-sets are generated by analyzing dependencies and other relations in order to identify other elements related to items in a given commit. Such a process allows linking of program elements at various levels of abstraction. Given a user query, related commits are identified based on the textual relationship of the user query and comments of the commits, and the associated set of program elements is augmented using the concept of transitivity which is subsequently returned to the user.

5. FEATURE LOCATION TOOLS AND STUDIES

In addition to the many research articles that introduce feature location techniques, there are numerous articles describing feature location tools, case studies, industrial studies, and user studies. This section summarizes these tools and studies.

[25] http://people.cs.ubc.ca/~murphy/AVID/ (accessed and verified on 03/01/2011)

5.1. Tools

Tool support for feature location removes much of the manual burden associated with searching for a feature's program elements. In addition to providing an overview of existing feature location techniques, this survey also describes tools that can be used for feature location. Some of the techniques summarized in Section 4 have prototype tools that are not available; therefore they are not listed here. Also, some tools are not directly associated with any particular approach, but they can be used for feature location, to document features, or program exploration, so they are included here.

5.1.1. Tools for Dynamic Feature Location

TraceGraph [Lukoit'00] is a feature location tool that allows for the visualization of execution traces. As a software system is running, TraceGraph analyzes the execution and visualizes which program elements were invoked during a time interval. The visualization is essentially a matrix in which the rows represent program elements, the columns correspond to time intervals, and the cells indicate if the program elements were called during that time interval or not. Additionally, the first invocation of a program element is highlighted in the visualization. TraceGraph was evaluated on the Mosaic web browser as well as the Joint Surveillance Target Attack Radar Subsystem (Joint STARS), a proprietary system developed by Northop Grumman for the United States Air Force. The tool's visualization was useful for feature location because it emphasized the first time an element was called, which often corresponded to a feature being triggered. TraceGraph was also applied in an industrial case study on feature location [Simmons'06] where it was used for trace differencing and identifying code uniquely executed by a feature, and in another study on distributed simulation software [Wilde'02].

STRADA (**S**cenario-based **TRA**ce **D**etection and **A**nalysis) [Egyed'07] is a tool developed to help programmers uncover the mappings between features and code during testing, which is based on trace analysis research [Egyed'03, Egyed'04, '05a, Egyed'05b]. Given a set of test cases for a feature, STRADA observes the code that is executed during testing, initially identifying all the executed code as relevant to the feature. However, since not all of the invoked code actually pertains to the feature, STRADA analyzes the traces using logical constraints to exclude irrelevant program elements. The tool visualizes its knowledge of feature-to-code mappings in a matrix. It has been evaluated on ArgoUML[26], GanttProject[27], and a video-on-demand player[28].

Olszak and Jørgensen proposed the tool Featureous [Olszak'10] for locating feature implementation in legacy software. The tool is implemented as a plug-in for the NetBeans IDE and allows developers to specify a feature, a scenario, and to collect execution traces that exercise the feature. The execution traces are analyzed and results are presented using advanced visualization views. Furthermore, the developer can navigate through the traceability links established through dynamic analysis. From an experience with JHotDraw, Featureous is reported to be able to support both top-down and bottom-up comprehension strategies using its visualization views.

5.1.2. Tools for Static Feature Location

Ripples [Chen'01b] is a tool that implements the ASDG approach to feature location. The tool extracts an ASDG from C code and visualizes it for the programmer who can

[26] http://argouml.tigris.org/ (accessed and verified on 03/01/2011)
[27] http://www.ganttproject.biz/ (accessed and verified on 03/01/2011)
[28] http://peace.snu.ac.kr/dhkim/java/MPEG/ (accessed and verified on 03/01/2011)

mark relevant nodes. JRipples[29] [Buckner'05] is an Eclipse plug-in and its functionality is similar to Ripples's tool, except that it is used on Java systems and it does not provide the visualization component. Both tools can also be used for impact analysis and change propagation by tracking and monitoring the status of program elements.

Suade [Weigand-Warr'08], is an Eclipse plug-in that performs feature location using static analysis. It implements the topology analysis approach discussed in Section 4.3. Suade has been used in a case study comparing several program exploration tools [de Alwis'07], and it has also been directly compared to Dora [Hill'07], another static feature location technique.

5.1.3. Tools for Textual Feature Location

Google Eclipse Search[30] (GES) [Poshyvanyk'06c] is an Eclipse plug-in that facilitates efficient source code searching and browsing by integrating Google Desktop Search (GDS)[31] and Eclipse. GSD is an off-the-shelf component that uses information retrieval. It allows users to search for files on their desktops similar to the way they would search for information on the Internet via natural language queries. By integrating GDS with Eclipse, programmers can search source code in a similar fashion. One advantage of using GDS is it unobtrusively re-indexes the search space when the source code changes. In a preliminary evaluation on the Java system Violet[32], GES was show to produce accurate results. In addition, while compared against Eclipse file search functionality, GES is considerably faster in producing the results.

IRiSS [Poshyvanyk'05] and JIRiSS [Poshyvanyk'06b] are both tools for textual feature location. IRiSS implements information retrieval-based feature location as an add-on to MS Visual Studio .NET, while JIRiSS is an Eclipse plug-in. Both tools work like a development environment's built-in search functionality, but instead of only displaying the lines of code that match a query, those lines' corresponding classes and methods are also listed. This allows a programmer to sort the results by different levels of granularity and to visit the classes or methods with the most matches. Also, since IR is used, the results returned from a query can be ranked by their relevance. JIRiSS is an extension to IRiSS that also includes fragment-based searches, software vocabulary extraction, query spell checking, and word suggestions to improve queries.

Xie et al. [Xie'06] developed a tool that supports textual analysis through visualization, by combining IRiSS [Poshyvanyk'05] and sv3D [Marcus'03]. IRiSS performs feature location via IR and sv3D (source viewer 3D) creates 3D renderings of the results, showing poly-cylinders that represent program elements. The colors of the poly-cylinders correspond to the elements' similarity to the query following a pre-defined color scheme. The height of the poly-cylinders represent browsing history, so the taller the cylinder, the more times the program element was visited in the past. The combination of these two tools allows a developer to have a visual representation of the results, as opposed to examine a ranked list of results.

Cleary and Exton implemented an Eclipse plug-in that supports the cognitive assignment technique [Cleary'06]. The tool allows a developer that is unfamiliar with a system to generate and store a set of links between the problem domain concepts stored in a cognitive map and the relevant parts of the source code. The developer can select a concept (i.e., feature) which wants to investigate, and based on the textual description of the feature the tool provides a set of results which the developer should investigate, and

[29] http://jripples.sourceforge.net/ (accessed and verified on 03/01/2011)
[30] http://ges.sourceforge.net/ (accessed and verified on 03/01/2011)
[31] http://desktop.google.com/ (accessed and verified on 03/01/2011)
[32] http://www.horstmann.com/violet/ (accessed and verified on 03/01/2011)

add them to the concept if they are relevant. The program elements suggested by the tool are the result of an analysis of links between words, using a Bayesian classifier.

5.1.4. Other Tools for Feature Location

In this subsection we discuss several tools that use multiple types of information in their analysis. Some of the tools use only a particular type of information, and they could be presented in different sections, but because they are highly related to one another, we present them in this section. For example, FEAT uses static analysis, and ConcernMapper uses textual analysis, but we present these tools in this subsection because these tools are highly related.

The motivation behind the following tools is that once a feature's relevant program elements have been found using feature location, they should be saved so that they could be easily accessed in the future. Features and their relevant program elements can be documented in Concern Graphs [Robillard'02, '07a], a model that describes which program elements pertain to a feature. An Eclipse plug-in that supports the Concern Graphs approach is FEAT[33] (**F**eature **E**xploration and **A**nalysis **T**ool) [Robillard'03b]. ConcernMapper [Robillard'05b] is also an Eclipse plug-in that supports Concern Graphs, and it evolved from FEAT. One of the improvements of ConcernMapper over FEAT is that it supports a fuzzy and less rigid model for representing concerns, which is more appropriate for programmers. ConcernTagger[34] is another Eclipse plug-in build on top of ConcernMapper, which provides the ability to compute a number of concern-specific metrics. The **F**eature **L**ocation and **T**extual **T**racing **T**ool[35] (FLAT^3) [Savage'10b] also extends ConcernMapper by adding support for the textual and dynamic feature location technique namely SITIR [Liu'07]. In each of these tools, programmers can define and name features and then associate entire or partial classes, methods, and fields with them. The tools leave feature location as a manual task and focus on documenting the features and their related elements once they are found. However, once the features and their program elements are documented, they can be saved and retrieved at a later time, thus avoiding the need to repeat searches.

Savage et al. [Savage'10a] developed TopicXP, an Eclipse plug-in that supports developers during maintenance tasks by analyzing the unstructured information embedded in the source code identifiers and comments using an advanced information retrieval technique, that is Latent Dirichlet Allocation [Blei'03]. LDA extracts a set of topics from the source code, which could be considered as concepts or features. The topics generated are mapped to the source code, and the relationship between the topics is determined by examining the static dependencies from the code. Using an interactive visualization view, the developer is able to navigate through these topics or to access the source code associated with them. In a preliminary study involving four graduate students, who were required to perform concept location on two Java systems, jEdit and muCommander[36], using Eclipse with TopicXP or using the just the Eclipse IDE. The results showed that TopicXP is a functional tool, which provides comparable results to Eclipse IDE, and in some cases it even produces better results.

Bohnet and Döellner [Bohnet'06a, b, '07a, b, '08a] visually explore dynamically extracted information, but in this case, as a call graph. Since a call graph can be large, in order to reduce the search space for the user, the tools provide clues to identify code relevant to the feature of interest. The tools also provide a number of different types of

[33] http://www.cs.mcgill.ca/~swevo/feat/ (accessed and verified on 03/01/2011)
[34] http://www.cs.columbia.edu/~eaddy/concerntagger/ (accessed and verified on 03/01/2011)
[35] http://www.cs.wm.edu/semeru/flat3/ (accessed and verified on 03/01/2011)
[36] http://www.mucommander.com/ (accessed and verified on 03/01/2011)

visualizations. In one prototype, the most important view, the graph exploration view, shows other methods that pass control flow to or receive control flow from a given method. In this view, the tool only shows methods in a neighborhood if they are judged to be relevant based on execution time, while another tool has textual and 3D landscape views. These tools effectively extract dynamic call graph information and guide programmers during navigation.

5.1.5. Other Tools not Included in the Survey

Program exploration tools support developers when performing a variety of maintenance tasks. Since feature location is central to many maintenance activities, we mention some of the program exploration tools which are somewhat out of scope of this survey.

JQuery [Janzen'03], an Eclipse plug-in, is a source code browsing tool designed to help programmers when dealing with features that have scattered implementations. The tool combines navigation based on relationships (as in a hierarchical browser) with the flexibility of query languages. Program exploration in JQuery begins with a query and a list of variables. The query determines which elements to show in the browser, and the variables establish how to organize them into a tree. The query defines the type of program element to search for given some parameters such as its name or a type of relationship. The results of the query are returned in a hierarchical tree, and users can further explore the tree with additional queries that expand nodes into sub-trees. The tool aims to reduce the burden of program investigation on developers. It helps them remain oriented by not having to switch between multiple views, and it records their exploration path in the tree format.

Ferret [de Alwis'08] is a tool for answering conceptual queries, which are questions about a software system a programmer may have. The Ferret model is based on the composition and integration of different sources of information into a query-able knowledge-base. A source of information is known as a sphere, and examples include static, structural relationships in source code, dynamic call information from an execution trace, and revision history recorded in a software repository. Ferret supports 36 different conceptual queries such as "What calls this method?", "What are this class' subclasses?", "What are all the fields declared by this type?", and "What transactions changed this element?" These types of queries represent questions programmers may have when investigating a software system in order to locate a feature's implementation.

Instead of relying on dynamic information, AspectBrowser [Shonle'04] is a tool that assumes that features follow the idea of information transparency [Griswold'01]; design decisions that cannot be encapsulated in a single module use a common signature or terminology that can easily be exploited by search tools. The AspectBrowser tool[37] allows users to search a code base using pattern matching and then visualizes the results in two ways. All query matches can be highlighted in the source code, and the programmer can browse to find them. Alternatively, programmers can use a global view to see how a feature is scattered throughout the system. In the view, each line of code is represented by a row of pixels, and highlighted rows indicate lines of code that match the query. Multiple search results can be viewed at once to understand the interaction between several features.

5.2. Case Studies

A number of case studies involving feature location have been performed, ranging from comparisons of existing techniques, industrial case studies, and user studies. Each type of the study is valuable to advance this research area. Comparisons evaluate few feature

[37] http://cseweb.ucsd.edu/~wgg/Software/AB/ (accessed and verified on 03/01/2011)

location techniques on the same systems and features, making it easier for researchers and practitioners to understand the strengths and weaknesses of each approach. Industrial case studies show the applicability of an approach in non-trivial settings. Finally, user studies provide insight into how programmers understand and search for code, and these insights can be incorporated into feature location techniques and tools.

5.2.1. Comparison of Feature Location Techniques using Case Studies

RECON, RECON2, and RECON3 are tools that implement the software reconnaissance approach to feature location. Wilde and Casey [Wilde'96] report on applying RECON to industrial software. In their study on using software reconnaissance for program exploration, Wilde and Casey found the tool to be very selective because it never marked more than 13 methods for a feature. They also observed that the tool seemed to find code that was near relevant program elements. In the second part of their study, they examined using software reconnaissance for traceability to build a large mapping of multiple features to code. The tool was used to run a large set of test cases that were marked with the features they exhibited, and then the collected traces were analyzed to find traceability relations that mapped features to code. With this traceability knowledge, a programmer modifying a program element is aware of the other features implemented in that program element.

When a new feature location technique is introduced, it is often directly compared with similar existing approaches as part of its evaluation. Some articles related to feature location focus solely on case studies comparing several techniques. Wilde et al. [Wilde'01, Wilde'03] compare software reconnaissance, ASDG, and grep in a case study to locate two features in legacy Fortran code. The system, CONVERT3, is part of a suite of geometric modeling programs and is used to convert models to formats required by other tools. For the study, three teams each used one of the feature location techniques to find the code for two features of CONVERT3. The software reconnaissance and ASGD teams were able to gain sufficient understanding of the source code, but the team using grep was not. The authors concluded that grep was the least reliable approach but it is very quick and can locate features that cannot be explicitly invoked dynamically. After grep, software reconnaissance was deemed to be the next fasted method of feature location. However, its results may not present a user with enough context to be comprehensible. The ASDG approach was the most difficult to apply but the most systematic and allows for the best understanding of the relevant code.

Ibrahim et al. [Ibrahim'03] also report on their experiences applying RECON2 the Generate Index (GI) project. Their findings echo the conclusions of the previous study. Software reconnaissance is based on test cases, but selecting appropriate scenarios to execute can be difficult. However, only a few test cases are generally needed for a feature. After the analysis, software reconnaissance is good at locating a starting point for feature location, but further investigation for additional relevant program elements should be performed.

Early feature location techniques were applied in the era of procedural programming paradigm. After object-oriented programming gained popularity, Marcus et al. [Marcus'05c] studied whether feature location was still needed since object-oriented code is supposed to be structured such that classes help implement well-defined problem domain concepts. They compared the performance of three static feature location techniques: pattern matching with grep, a depth-first dependency search, and information retrieval using LSI. Three programmers, each assigned to a different technique,

participated in a case study to locate features in Art of Illusion[38], a 3D modeling studio written in Java, and in Doxygen[39], a source code documentation generator written in C++. They concluded that object-orientation does not always allow for quick and easy identification of the program elements relevant to a feature. Therefore, feature location techniques are still needed for object-oriented systems.

Revelle and Poshyvanyk [Revelle'09] performed an exploratory study evaluating several feature location techniques that return ranked lists of program elements (methods). The approaches they compared were Information Retrieval (LSI-based feature location), Information Retrieval plus dynamic analysis (SITIR), and Information Retrieval plus dynamic and static analysis (similar to Cerberus). For IR, they assessed user-formulated queries as well as method seed queries in which the text of a method already known to be relevant to a feature was used as the query. For dynamic analysis, they used both full execution traces and marked traces in which only the portion of a system's execution when a feature is invoked was traced. Dynamic analysis was combined with IR by pruning unexecuted methods from the ranked list. When all three types of analyses were combined in IR + Dyn + Static, a program dependence graph was traversed starting from a seed by following dependencies only if they were executed and had textual similarities to the query that were above a given threshold. Most feature location techniques that return a ranked list are evaluated in terms of where the first relevant element appears on the list. This case study aimed to evaluate these approaches in terms of how well they find near-complete implementations of features, meaning how well they find as many relevant program elements as possible. Their conclusions were that none of these approaches perform particularly well in that regard since feature location is usually used to find a starting point and impact analysis tools are used to find more complete implementations. They observed that marked traces generally outperformed full traces and that the method seed queries, which can be automatically generated, performed just as well as user formulated queries.

De Alwis et al. [de Alwis'07] performed a comparative study in which programmers used three tools (JQuery, Ferret, and Suade) to plan complex maintenance tasks. Eclipse was used as a baseline for comparison. They hypothesized that programmers (i) would find it easier to complete a task using a tool as opposed to Eclipse, (ii) need to examine less code as compared to using Eclipse, and (iii) would identify more important elements using the tool as opposed to Eclipse. The participants in the study were 18 professional programmers, and they were asked to investigate two change tasks in jEdit. In the first task, they used only Eclipse, and in the second task, they used one of the exploration tools. The order of the tasks and choice of tools was randomized. An instrumented version of Eclipse captured all events the programmers performed during their investigation. Additionally, the participants recorded the relevant elements they found in an Eclipse view built for the study. The NASA Talk Load Index (TLX) was used to assess task difficulty, and distance profiles were used to gauge the degree to which the participants remained on-task. The TLX scores showed no difference in task difficulty that could be attributed to using a tool or not. Similarly, the distance profiles did not indicate that the tools had any strong effect on the tasks. Overall, the authors concluded that program exploration tools had little effect, and that the behavior of the developers seemed more impacted by the tasks performed. In addition, there was evidence that individual programmers' strategies caused them to be more or less efficient.

[38] http://aoi.sourceforge.net (accessed and verified on 03/01/2011)
[39] http://www.stack.nl/~dimitri/doxygen/ (accessed and verified on 03/01/2011)

5.2.2. Industrial Case Studies

Most feature location case studies focus on open source software. However, case studies carried out on industrial software give a sense of a technique's real world applicability. Unfortunately, only a few such studies have been performed, and more are needed. As previously discussed, TraceGraph was used in an industrial setting [Lukoit'00], and Wilde et al. [Wilde'03] compared a number of approaches on industrial software. In addition to these studies, Van Geet and Demeyer [Van Geet'09] report on their experiences of applying Eisenbarth et al.'s [Eisenbarth'03] formal concept analysis of execution traces feature location technique in an industrial setting. The context was a pre-study phase for the migration of a banking system written in COBOL. Scenarios for two features were executed using a web interface, and three separate iterations of the approach were conducted. Each iteration aimed at reducing the number of modules considered by using different combinations of scenarios that did and did not invoke the feature. A domain expert provided the modules relevant to each feature for evaluation purposes, and in two out of three iterations of the approach, all the relevant modules were in the generated concept lattice. Three additional relevant modules were also identified that had not previously been named by the domain expert.

5.2.3. User Studies

Studies that focus on how programmers search and comprehend source code are important to feature location research. These types of studies provide insights on how developers find a feature's implementation or gain understanding of a system. In turn, these insights can be incorporated into feature location research in order to develop approaches that are organic and easy for programmers to use. Four relevant user studies are discussed below, and while this is not a complete list of user studies related to feature location, even more studies are necessary to advance the state of the art.

LaToza et al. [LaToza'07] performed a user study in which 13 participants worked for three hours on understanding and improving the design of two features in jEdit. The participants' activities were recorded using think-aloud, video, and Eclipse instrumentation. The goal of the study was to answer questions about how programmers' experience affects the changes they make to code, how it affects how they work, and how they reason about design during coding tasks. LaToza et al. found that the more experienced programmers addressed the causes of problems, while beginners focused on the symptoms and that the experienced programmers identified relevant methods and implemented changes more quickly than novices. They also discovered that the participants' activities centered on fact finding. The programmers sought facts relevant to their task, so they investigated certain methods and learned facts about the software system, and as they learned enough facts, they were able to propose design changes. Therefore, feature location techniques should not only help identify relevant program elements, but they should also aid in fact finding and program comprehension.

Robillard et al. [Robillard'04] also conducted a study on how programmers explore source code when performing a change task. Five programmers were asked to modify jEdit so that users can explicitly disable the auto-save functionality and given five requirements for their solution. The data collected included artifacts produced or modified by the participants as well as video recordings of their screens. The programmers' success was judged in terms of time to complete the task and the quality of their change to the source code in terms of how many of the task's requirements they successfully implemented. Robillard et al. analyzed the behavior of each participant by transcribing the screen videos into events. Each event records the time it occurred, the

method being examined at that time, how the method was accessed (scrolling, browsing, searching, etc), and whether the method was modified. Based on their observations, they conclude that a methodical, ordered investigation of a system's source code is more effective than a systematic, opportunistic one. They found that programmers should follow a plan when exploring a program, that they should perform focused searches in the context of their plan, and that they should keep record of their findings. Based on these results, feature location techniques should facilitate orderly program exploration.

Revelle et al. [Revelle'05] undertook an exploratory study on how programmers identify features and their implementations in source code. In the first study, the features of GNU sort[40] plus their relevant source code was found manually by one programmer and then compared to those of Carver and Griswold [Carver'99]. In the second study, two programmers manually located features and their implementations for a Java implementation of the Minesweeper game. Revelle et al. compared the actual concepts recognized as features as well as the code associated with those features, looking for common trends in how developers identify and locate features. Based on their observations, they developed a set of guidelines for how to identify and recognize the existence of a feature and how to record feature's associated code in a tool called Spotlight [Coppit'07]. The guidelines suggest relying on both static and textual information and flexible mappings of features to program elements of various levels of granularity.

Ko et al. [Ko'05, Ko'06] performed an exploratory study to investigate developers' strategies for understanding unfamiliar code. Ten participants worked using Eclipse on five maintenance tasks associated with the Paint[41] application. Screen-capture videos recorded the developers' work during the study. To simulate a more realistic working environment, the programmers were interrupted every 2.5 to 3.5 minutes and required to answer a multiplication question. Monetary incentives were offered for correctly completing the tasks, and penalties were inflicted for ignoring or incorrectly answering the multiplication questions. The study found that programmers interleave three activities when exploring source code: searching for relevant code either manually or with tools, following the dependencies of found relevant code, and collecting relevant code and information in Eclipse's interface (*i.e.*, package explorer, tabs, and scroll bars). However, searches often failed, Eclipse's navigation tools imposed overhead when following dependencies, and developers lost track of relevant code in the interface. On average, 35% of a developer's time was spent reviewing search results and on navigation. Based on the observations of this study, the authors make a number of suggestions for future tool development. First, tools need to provide better relevance cues so programmers do not miss important code or misinterpret irrelevant code. Second, dependency searches need to be more practical, such as by highlighting the dependencies of the currently selected program element. Third and finally, programmers need a better way to collect, organize and view the relevant information they find, such as being able to see all relevant information for a given task at once. These recommendations may help programmers find task-relevant code more quickly and efficiently and were used in the design of a new debugging tool [Ko'08].

6. ANALYSIS OF THE TAXONOMY

Using the taxonomy we are able to address the research questions:

RQ1: What types of analysis are used when performing feature location? Results of our systematic survey point out that feature location techniques primarily use dynamic,

[40] http://www.gnu.org/software/coreutils/ (accessed and verified on 03/01/2011)
[41] http://www.cs.cmu.edu/~marmalade/studies.html (accessed and verified on 03/01/2011)

static and textual information to locate features in source code. However, analysis techniques for feature location are not limited to dynamic, static, or textual analysis as techniques which combine various types of analysis as well as those which utilize history based analysis are also prevalent in the literature. For a detailed discussion of the various types of analysis see Section 3.1. Likewise, discussion classifying the various feature location techniques according to their respective type of analysis appears in Section 4.

RQ2: Has there been a change in types of analysis used to identify features in source code employed by recent feature location techniques? Our survey reveals that, in recent years, there has been an increase in the number of papers published that introduce feature location techniques based on textual information as well as historical data (see Section 4). This can be observed in Figure 3 (dotted line), which displays the cumulative number of feature location techniques that use different types of information per year. We choose this cumulative visual representation of approaches, as opposed to a direct scattering of the number of approaches per year because the graph is much easier to understand. In addition to the emergence of new analysis techniques our survey indicates that a growing number of researchers have demonstrated the benefits of combining multiple analysis techniques to leverage the complementary strengths of each analysis mechanism (see solid lines in Figure 3).

RQ3: Are there any limitations to current strategies for evaluating various feature location techniques? Two observations that can be made from our feature location survey include (1) limited comparison of proposed techniques with existing techniques and, more notably, (2) limited use of benchmarks in evaluations. Comparison of proposed approaches to other existing techniques should be common practice while introducing new feature location techniques. However, only 22 of the 58 papers (38%) that present FLTs (see Section 4, Table 3 and Table 4) compared their approaches with a limited number of existing approaches. In rare instances existing techniques may not be relevant to be used during evaluation. Nevertheless, researchers ought to compare their new techniques against those that appear in the literature. The apparent lack of comparison may be attributed to the low number of techniques which provide a corresponding publicly available tool. Without tools for each of the existing techniques researchers need to implement both the proposed technique as well as relevant techniques

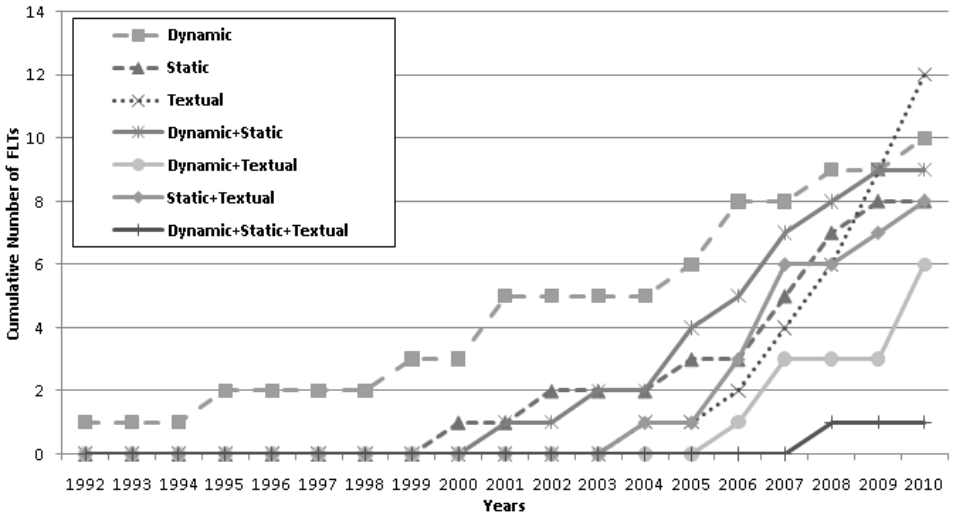

Figure 3 Cumulative number of FLTs from Table 3 and Table 4 per year

which should be compared against the proposed one. Such a process is time consuming and also introduces a threat to validity (possibility of erroneous implementation of existing feature location techniques). As it stands direct comparison of approaches are limited, however, may be increased if researchers produce publicly available tools. The second observed limitation is the lack of evaluation on benchmarks. In fact, only 3 of the 58 papers (5%) that present FLTs (see Section 4, Table 3 and Table 4) used benchmarks in their evaluation. If benchmarks were readily available for researchers to evaluate their approaches, availability of publicly available tools would not be such a significant problem, since researchers would have been able to directly compare their results to the results, which appear on the same datasets.

7. DISCUSSION AND OPEN ISSUES

Feature location is an essential aspect of many software maintenance tasks, and because it can be challenging to perform manually, researchers have introduced many techniques to lessen the burden of searching for a feature's relevant code. Even with these numerous approaches and advancements, open issues remain in the field of feature location. One question that is unanswered is, "*What is the best way to perform feature location?*" This question cannot be easily answered without an extensive comparison of analysis-specific issues and a comparison of existing approaches. Note that while addressing RQ_3 we observed that comparisons of existing approaches are simply missing from most evaluations in the feature location literature. This means that even though different approaches used in their evaluations the same systems (e.g., Eclipse, JHotDraw, jEdit, Mozilla, etc.), they did not use the same datasets (e.g., version, source code, gold set, etc.) for those systems. In addition, the approaches were not restricted to use the same evaluation metrics (e.g., precision, recall, effectiveness, etc.), which makes comparison between approaches even more difficult. Such a comparison could be facilitated by well-established benchmarks, but currently, there is no commonly accepted set of features associated with the code that implements them that could be used to compare feature location techniques. The issue of benchmarks was identified as we addressed RQ_3 using our systematic survey. Such a benchmark is needed in the research area. Additionally, while there are various techniques that support feature location, not all approaches have publicly available tools, and the tools that are available do not support both locating and documenting a feature's implementation. Other open issues are usability studies of feature location techniques and integrating feature location into software engineering courses. The remainder of this section discusses these open issues and their associated avenues for future research. While this discussion brings to light these important topics, more panels and workshops, like the one on the "Identifications of concepts, features, and concerns in source code" held at the 21st IEEE International Conference on Software Maintenance (ICSM2005), are necessary to resolve many of these issues.

7.1. Comparisons

Given a wide variety of existing techniques, developers that need to perform feature location have many options, but which approach is the best for a specific situation? What parameters should be used for a certain type of analysis? Which type of analysis yields the best results, or is a combination of analyses the best? Some case studies have been performed comparing multiple feature location techniques, but they only have a few data points, which impede the ability to draw statistically significant generalizations from their results. These studies are also limited in the number of examined approaches, focusing on a subset of approaches that present results in a similar fashion. An obstacle to comparing techniques is the presentation of their results. How does one evaluate one result set that

ranks program elements to another that does not? Determining the best way to directly compare the performance of feature location techniques remains an open issue.

Not only does there need to be a comparison of techniques based on different types of analysis, but there also needs to be an evaluation of the best configuration of each type of analysis. For instance, dynamic analysis has many possible options for collecting traces. The granularity of execution traces can be classes, methods, or even lines of code. In addition, the entire execution can be logged from start up to shut down, or only select portions of the run can be captured. With static analysis, like with dynamic, granularity is also a parameter. Additionally, the type of dependencies (control or data) to take into account is another issue to consider. With textual analysis, preprocessing options, such as stemming and stop word removal, are commonly used, but their effect on feature location has not been fully studied. Also, textual analysis can be achieved through Information Retrieval methods or through natural language processing. While the varied IR methods have been compared, the effectiveness of IR and NLP has not been compared in the context of feature location. A comparison would determine if the extra expense associated with NLP is worth the precision, or if the less expensive IR methods are sufficient. Thorough investigations comparing these different configurations of each type of analysis would reveal the most favorable settings for feature location.

There are many other open feature location issues that could potentially be resolved through comparisons. The main types of analyses are dynamic, static, and textual; although historical analysis has also been used, but not in conjunction with any other analysis. It remains to be seen if combining historical analysis with any of the others is a viable approach to feature location. Just as three types of analysis comprise the majority of exiting techniques, two programming languages dominate the area of feature location. Most existing approaches have been applied to Java or C/C++. However, feature location should branch out to support more programming languages.

7.2. Benchmarks

The comparison of feature location techniques should be facilitated by the existence of benchmarks that could be used to consistently evaluate the approaches. Currently, there are a number of systems that have been used in the evaluation of many feature location techniques such as Eclipse, JHotDraw, jEdit, Mozilla, and Firefox, but sets of features used for the evaluation are not consistent. Even if two approaches are evaluated on the same system, if different features are used, comparing two techniques is difficult. Another problem with assessing feature location is in knowing the “gold set” of program elements that implement a feature. The most commonly used method for determining the source code relevant to a feature is to mine bug tracking systems. However, the code associated with a feature may be incomplete if a bug fix only touches part of a feature’s implementation. In the presence of these issues, the field of feature location research needs to establish solid standards for validation. The best solution may be benchmarks that can facilitate easy comparison of FLT approaches. The benchmarks should include a set of features from a software system or several systems. Each feature needs to be mapped to the source code implementing that. Preferably, the benchmarks should be defined at various granularities to support different approaches that identify relevant classes, methods, or statements. Robillard et al. [Robillard'07b], Eaddy et al. [Eaddy'08b], Revelle et al. [Revelle'10] took a step in this direction, making their data sets available in which programmers, who were not necessarily systems experts, mapped features to classes, methods, and fields in open source Java applications. Still, well-established and complete benchmarks for systems from a variety of domains and languages will make the evaluation and comparison of feature location techniques easier.

In this survey, we make a step towards solving this problem by making a set of benchmarks publicly available for several systems that were used frequently in case studies. The purpose of making these datasets available to researchers is twofold. First, we want to facilitate the evaluation of new approaches on larger and diverse datasets, in order to make the approach more generalizable. Second, we want to ease the comparison between different FLTs. Using the same datasets, a new approach can be straightforwardly compared against an existing approach, which could provide a great indication about the value of the proposed technique. We were encouraged to make available the benchmark by observing the success of Eaddy et al.'s [Eaddy'08b] benchmark. The Rhino dataset from that benchmark was used in several evaluations and even by different research groups as well [Eaddy'08a, Eaddy'08b, Hill'09, Revelle'10].

The benchmarks that we make available could be downloaded from the website http://www.cs.wm.edu/semeru/data/benchmarks/. The website contains detailed instructions about the format of the datasets and the process used to generate them. These benchmarks include gold sets (i.e., mappings between source code and features) at method level granularity, description of the features from bug reports and in some cases even execution traces. In the near future, we plan to generate more benchmarks and add them to this website, but for now we are making available the following datasets.

The first benchmark is for jEdit version 4.3 and it contains 272 features that have associated gold sets and issues from the issue tracking system, which can be used to extract the textual descriptions of the features/bugs. In addition, 150 features have associated execution traces that were generated by following the steps to reproduce enumerated in the issue description. The gold sets were generated via analysis of the SVN commits submitted between releases 4.2 and 4.3. The issue identifiers from the SVN logs were extracted, and were mapped to issues from the issue tracking system. On the other hand, the changes from the SVN commits were mapped to the methods from the source code that were modified by that commit.

The second benchmark we make available is for ArgoUML version 0.22 and it contains 132 features that have gold sets and issues from the issue tracking system respectively. In addition, 91 features have associated execution traces that were generated by following the steps to reproduce described in the issue description. The process of generating the gold sets is similar to the one used for generating the benchmark for jEdit system.

The third benchmark we make available is for muCommander version 0.8.5 and it contains 92 features that have gold sets, issues from the issue tracking system as well as execution traces. The process of gathering the data for this benchmark is similar to ArgoUML's process.

The fourth benchmark is for JabRef[42] version 2.6 and it contains 39 features that have associated gold sets, issues as well as execution traces. The data collection process is similar to ArgoUML's.

The fifth benchmark we provide is for Eclipse version 3.0 and it contains 45 features represented by bug reports submitted to Eclipse's issue tracking system. Unlike for the jEdit and ArgoUML datasets, the gold sets were generated by examining the patches submitted in the bug reports, which contain information about the code that was changed to fix the bug. We also provide 45 execution traces with this dataset, which were collected by following the steps to reproduce from the description of the bugs. The Eclipse benchmark was used in the evaluation by Revelle et al. [Revelle'10].

7.3. Tools

[42] http://jabref.sourceforge.net/ (accessed and verified on 03/01/2011)

Even though this survey encompasses many tools that support feature location (see Section 5.1), the majority of feature location techniques do not have a publicly available tool, meaning that programmers wanting to apply such an approach may need to recreate the technique's methodology. Additionally, some tools are useful for investigating a program and locating features, while others can be used to store the mappings between features and source code, but there are currently no tools that perform both. Combining the functionalities of finding features' implementations and being able to save them is a logical next step for tool development. Finally, de Alwis et al.[de Alwis'07] found that existing tools have little effect on programmer's efficiency, so further research needs to be done to improve usability of the tools.

7.4. User Studies

While there have been several user studies investigating how programmers search and explore source code during maintenance, these studies are conducted with a relatively small number of developers. Further studies are needed with more programmers to be able to derive conclusive and statistically significant results. Additionally, there has been a lack of empirical studies examining usability aspects of feature location. *Do existing feature location techniques reduce the amount of time and effort developers spend on maintenance? What are the practical benefits and costs of using different types of approaches?* For instance, collecting execution traces for an approach that uses dynamic analysis requires overhead in terms of the time spent to develop scenarios or test cases and capture traces. Information Retrieval involves indexing the source code, which can be rather time-consuming for large software systems. Studies are needed to determine whether or not the overhead of collecting traces or (re)indexing a corpus yields improved feature location results and is worth the cost.

7.5. Feature Location and Education

Given that feature location is such an extensive area of research and also an important part of software maintenance, it should be taught in software engineering courses at universities and colleges. Petrenko et al. [Buchta'06, Petrenko'07] have argued for the inclusion of software maintenance and evolution in software engineering courses along with traditional development. Teaching maintenance exposes students to more realistic experiences since in industry, 70% or more of programmers' time is devoted to maintenance [Schach'01, Somerville'01]. Feature location is a significant part of the maintenance phase because before changes can be made to a system, the relevant program elements must be found. Therefore, feature location should be introduced as a topic in software engineering courses to better prepare students.

8. CONCLUSION

Through a comprehensive examination of 89 feature location articles encompassing research, tool, and case, industrial, and user studies, this survey has presented a taxonomy that classifies the literature along nine key dimensions. The taxonomy facilitates the comparison of existing feature location techniques and illuminates possible areas of future research. Researchers can use the taxonomy and survey as a basis for advancing the field, while practitioners can use it to identify techniques and tools that are well-suited to their needs. This survey has also shed light on open issues in feature location, such as the need for comparisons and benchmarks. By structuring the research area of feature location, this taxonomy and survey contribute well-defined organization to the field and should aid in resolving some of the open issues. This systematic survey should

serve not only academic researchers but also industrial professionals, aiming at adopting feature location tools within their organizations and development processes.

9. ACKNOWLEDGEMENTS

We are grateful to the anonymous JSME reviewers for their relevant and useful comments and suggestions, which helped us in significantly improving the earlier versions of this taxonomy, the survey and the paper in general. We would also like to thank Andrian Marcus for his useful and stimulating suggestions on the earlier drafts of this survey paper. We would also like to express our gratitude to some of the authors whose work is discussed in this survey for their insightful feedback in preparing the camera ready version of this paper. This work is supported by NSF CCF-0916260 and NSF CCF-1016868 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[Abebe'10] Abebe, S. L. and Tonella, P., (2010), "Natural Language Parsing of Program Element Names for Concept Extraction", in Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2, pp. 156-159.

[Agrawal'98] Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Ghosh, S., and Wilde, N., (1998), "Mining System Tests to Aid Software Maintenance", *Computer*, vol. 31, no. 7, July 1998, pp. 64-73.

[Antoniol'05] Antoniol, G. and Guéhéneuc, Y., (2005), "Feature Identification: A Novel Approach and a Case Study", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25, pp. 357-366.

[Antoniol'06] Antoniol, G. and Guéhéneuc, Y. G., (2006), "Feature Identification: An Epidemiological Metaphor", *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 627-641.

[Asadi'10] Asadi, F., Di Penta, M., Antoniol, G., and Guéhéneuc, Y. G., (2010), "A Heuristic-based Approach to Identify Concepts in Execution Traces", in Proceedings of 14th European Conference on Software Maintenance and Reengineering (CSMR'10), Madrid, Spain, March.

[Biggerstaff'94] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., (1994), "The Concept Assignment Problem in Program Understanding", in Proceedings of 15th IEEE/ACM International Conference on Software Engineering (ICSE'94) May 17-21, pp. 482-498.

[Binkley'10a] Binkley, D. and Lawrie, D., (2010a), "Information Retrieval Applications in Software Development", in *Encyclopedia of Software Engineering*, P. Laplante, Ed.: Taylor & Francis LLC.

[Binkley'10b] Binkley, D. and Lawrie, D., (2010b), "Information Retrieval Applications in Software Maintenance and Evolution", in *Encyclopedia of Software Engineering*, P. Laplante, Ed.: Taylor & Francis LLC.

[Blei'03] Blei, D. M., Ng, A. Y., and Jordan, M. I., (2003), "Latent Dirichlet Allocation", *Journal of Machine Learning Research*, vol. 3, pp. 993-1022.

[Bohnet'06a] Bohnet, J. and Döellner, J., (2006a), "Analyzing Feature Implementation by Visual Exploration of Architecturally-Embedded Call-Graphs", in Proceedings of International Workshop on Dynamic Systems Analysis (WODA'06), Shanghai, China, pp. 41-48.

[Bohnet'06b] Bohnet, J. and Döellner, J., (2006b), "Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems", in *ACM Symposium on Software Visualization (SOFTVIS'06)*. Brighton, United Kingdom, pp. 95-104.

[Bohnet'07a] Bohnet, J. and Döellner, J., (2007a), "CGA Call Graph Analyzer - Locating and Understanding Functionality within the Gnu Compiler Collection's Million Lines of Code", in Proceedings of 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07), Banff, Alberta, Canada, June 25-26, pp. 161-162.

[Bohnet'07b] Bohnet, J. and Döellner, J., (2007b), "Facilitating Exploration of Unfamiliar Source Code by Providing 2½D Visualizations of Dynamic Call Graphs", in Proceedings of 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07), Banff, Alberta, Canada, June 25-26, pp. 63-66.

[Bohnet'08a] Bohnet, J. and Döellner, J., (2008a), "Analyzing Dynamic Call Graphs Enhanced with Program State Information for Feature Location and Understanding", in Proceedings of 30th IEEE/ACM International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 10-18, pp. 915-916.

[Bohnet'08b] Bohnet, J., Voigt, S., and Dollner, J., (2008b), "Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-Focused Views on Execution Traces", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, June 10-13, pp. 268-271.

[Brin'98] Brin, S. and Page, L., (1998), "The Anatomy of a Large-Scale Hypertextual Web Search Engine", in Proceedings of 7th International Conference on World Wide Web, Brisbane, Australia, pp. 107-117.

[Buchta'06] Buchta, J., Petrenko, M., Poshyvanyk, D., and Rajlich, V., (2006), "Teaching Evolution of Open Source Projects in Software Engineering Courses", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), pp. 136-144.

[Buckner'05] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., (2005), "JRipples: A Tool for Program Comprehension during Incremental Change", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, USA, May 15-16, pp. 149-152.

[Carver'99] Carver, L. and Griswold, W. G., (1999), "Sorting Out Concerns", in Proceedings of OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns.

[Chen'01a] Chen, A., Chou, E., Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., and Michail, A., (2001a), "CVSSearch: searching through source code using CVS comments", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'01), Nov., pp. 364-373.

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

[Chen'01b] Chen, K. and Rajlich, V., (2001b), "RIPPLES: Tool for Change in Legacy Software", in Proceedings of International Conference on Software Maintenance (ICSM'01), Florence, Italy, November 07 - 09, 2001, pp. 230-239.

[Cleary'06] Cleary, B. and Exton, C., (2006), "The Cognitive Assignment Eclipse Plug-in", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, June 14-17, pp. 241-244.

[Cleary'07] Cleary, B. and Exton, C., (2007), "Assisting Concept Location in Software Comprehension", in Proceedings of 19th Psychology of Programming Workshop, pp. 42-55.

[Cleary'09] Cleary, B., Exton, C., Buckley, J., and English, M., (2009), "An empirical analysis of information retrieval based concept location techniques in software comprehension", *Empirical Software Engineering*, vol. 14, no. 1, pp. 93-130.

[Comon'94] Comon, P., (1994), "Independent component analysis, a new concept?", *Signal Process.*, vol. 36, no. 3, Apr. 1994, pp. 287-314.

[Coppit'07] Coppit, D., Painter, R., and Revelle, M., (2007), "Spotlight: A Prototype Tool for Software Plans", in Proceedings of International Conference on Software Engineering (ICSE'07), pp. 754-757.

[Cornelissen'10] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R., (2010), "A Systematic Survey of Program Comprehension through Dynamic Analysis", *IEEE Trans. on Software Engineering*.

[Cubranic'03] Cubranic, D. and Murphy, G. C., (2003), "Hipikat: Recommending pertinent software development artifacts", in Proceedings of 25th International

Conference on Software Engineering (ICSE'03), Portland, OR, May, pp. 408-418.

[Cubranic'04] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., (2004), "Learning from project history: a case study for software development", in Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW'04), Chicago, Illinois, USA, pp. 82-91.

[Cubranic'05] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., (2005), "Hipikat: A Project Memory for Software Development", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June, pp. 446-465.

[de Alwis'08] de Alwis, B. and Murphy, G. C., (2008), "Answering conceptual queries with Ferret", in Proceedings of Proceedings of the International Conference on Software Engineering (ICSE'08), Leipzig, Germany, pp. 21-30.

[de Alwis'07] de Alwis, B., Murphy, G. C., and Robillard, M., (2007), "A Comparative Study of Three Program Exploration Tools", in Proceedings of 15th IEEE International Conference on Program Comprehension, pp. 103-112.

[Deerwester'90] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., (1990), "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, pp. 391-407.

[Dufour'07] Dufour, B., Ryder, B. G., and Sevitsky, G., (2007), "Blended Analysis for Performance Understanding of Framework-based Applications", in Proceedings of International Symposium on Software Testing and Analysis (ISSTA'07), London, United Kingdom, pp. 118-128.

[Eaddy'08a] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y. G., (2008a), "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, pp. 53-62.

[Eaddy'08b] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., and Aho, A. V., (2008b), "Do Crosscutting Concerns Cause Defects?", *IEEE Transaction on Software Engineering*, vol. 34, no. 4, July-August, pp. 497-515.

[Edwards'06] Edwards, D., Simmons, S., and Wilde, N., (2006), "An approach to feature location in distributed systems", *Journal of Systems and Software*, vol. 79, no. 1, Jan. 2006, pp. 57-68.

[Edwards'09] Edwards, D., Wilde, N., Simmons, S., and Golden, E., (2009), "Instrumenting Time-Sensitive Software for Feature Location", in Proceedings of International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, pp. 130-137.

[Egyed'03] Egyed, A., (2003), "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions on Software Engineering (TSE)*, vol. 29, no. 2, February, pp. 116 - 132.

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

[Egyed'04] Egyed, A. and Grünbacher, P., (2004), "Identifying Requirements Conflicts and Cooperation", *IEEE Software*, vol. 21, no. 6, Nov. 2004, pp. 50-58.

[Egyed'05a] Egyed, A. and Grünbacher, P., (2005a), "Supporting SOftware Understanding with Automated Traceability", *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 5, Oct. 2005, pp. 783-810.

[Egyed'05b] Egyed, A., Heindl, M., Biffl, S., and Grünbacher, P., (2005b), "Determining the Cost-Quality Trade-off for Automated Software Traceability", in Proceedings of International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, pp. 360-363.

[Eisenbarth'01a] Eisenbarth, T., Koschke, R., and Simon, D., (2001a), "Aiding Program Comprehension by Static and Dynamic Feature Analysis", in Proceedings of International Conference on Software Maintenance (ICSM01), Florence, Italy, November 7-9, pp. 602-611.

[Eisenbarth'01b] Eisenbarth, T., Koschke, R., and Simon, D., (2001b), "Derivation of Feature Component Maps by means of Concept Analysis", in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'01), pp. 176-179.

[Eisenbarth'01c] Eisenbarth, T., Koschke, R., and Simon, D., (2001c), "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces", in Proceedings of International Workshop on Program Comprehension (IWPC'01), pp. 300-309.

[Eisenbarth'03] Eisenbarth, T., Koschke, R., and Simon, D., (2003), "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March, pp. 210 - 224.

[Eisenberg'05] Eisenberg, A. D. and De Volder, K., (2005), "Dynamic Feature Traces: Finding Features in Unfamiliar Code", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30, pp. 337-346.

[Ernst'03] Ernst, M., (2003), "Static and Dynamic Analysis: Synergy and Duality", in Proceedings of ICSE Workshop on Dynamic Analysis (WODA'03), Portland, OR, May, pp. 24-27.

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

[Furnas'87] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T., (1987), "The Vocabulary Problem in Human-System Communication", *Communications of the ACM*, vol. 30, no. 11, pp. 964-971.

[Gao'04] Gao, J., Nie, J.-Y., Wu, G., and Cao, G., (2004), "Dependence Language Model for Information Retrieval", in Proceedings of International ACM SIGIR Conference on Research and Development in Information Retrieval, Sheffield, United Kingdom, pp. 170-177.

[Gay'09] Gay, G., Haiduc, S., Marcus, M., and Menzies, T., (2009), "On the Use of Relevance Feedback in IR-Based Concept Location", in Proceedings of 25th IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada, September, pp. 351-360.

[Grant'08] Grant, S., Cordy, J. R., and Skillicorn, D. B., (2008), "Automated Concept Location Using Independent Component Analysis ", in Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, October 15-18, pp. 138-142.

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

[Greevy'06] Greevy, O., Ducasse, S., and Girba, T., (2006), "Analyzing software evolution through feature views", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 6, November, pp. 425 - 456.

[Griswold'01] Griswold, W. G., (2001), "Coping with Crosscutting Software Changes Using Information Transparency", in Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September.

[Hayashi'10a] Hayashi, S., Sekine, K., and Saeki, M., (2010a), "iFL: An interactive environment for understanding feature implementations", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, Sept 12 -18, pp. 1-5.

[Hayashi'10b] Hayashi, S., Yoshikawa, T., and Saeki, M., (2010b), "Sentence-to-Code Traceability Recovery with Domain Ontologies", in Proceedings of 17th Asia-Pacific Software Engineering Conference (APSEC'10), Sydney, Australia, Nov 30 - Dec 3, pp. 385-394.

[Hill'07] Hill, E., Pollock, L., and Vijay-Shanker, K., (2007), "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), November, pp. 14-23.

[Hill'09] Hill, E., Pollock, L., and Vijay-Shanker, K., (2009), "Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse", in Proceedings of 31st IEEE/ACM International Conference on Software Engineering (ICSE'09), May 16-24.

[Ibrahim'03] Ibrahim, S., Idris, N. B., and Deraman, A., (2003), "Case study: Reconnaissance techniques to support feature location using RECON2", in

Proceedings of Asia-Pacific Software Engineering Conference (APSEC'03), pp. 371-378.

[Janzen'03] Janzen, D. and Volder, K., (2003), "Navigating and querying code without getting lost", in Proceedings of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), pp. 178-187.

[Kitchenham'04] Kitchenham, B., (2004), "Procedures for Performing Systematic Reviews", vol. TR/SE-0401, I. 1353-7776, Ed. Keele, UK: Keele University.

[Kleinberg'99] Kleinberg, J. M., (1999), "Authoritative Sources in a Hyperlinked Environment", *Journal of the ACM*, vol. 46, no. 5, pp. 604-632.

[Ko'05] Ko, A. J., Aung, H. H., and Myers, B. A., (2005), "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks", in Proceedings of 27th IEEE/ACM International Conference on Software Engineering (ICSE'05), May 15-21, pp. 126-135.

[Ko'08] Ko, A. J. and Meyers, B. A., (2008), "Debugging reinvented: asking and answering why and why not questions about program behavior", in Proceedings of International Conference on Software Engineering (ICSE'08), Leipzig, Germany, pp. 301-310.

[Ko'06] Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H., (2006), "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, December, pp. 971-987.

[Koschke'05] Koschke, R. and Quante, J., (2005), "On dynamic feature location", in Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, CA, USA, pp. 86-95.

[Kothari'06] Kothari, J., Denton, T., Mancoridis, S., and Shokoufandeh, A., (2006), "On Computing the Canonical Features of Software Systems", in *13th IEEE Working Conference on Reverse Engineering (WCRE'06)*. Benevento, Italy.

[LaToza'07] LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A., (2007), "Program Comprehension as Fact Finding", in Proceedings of The 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering, pp. 361 - 370.

[Liu'07] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., (2007), "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, Georgia, November 5-9, pp. 234-243.

[Lukins'08] Lukins, S., Kraft, N., and Etzkorn, L., (2008), "Source Code Retrieval for Bug Location Using Latent Dirichlet Allocation", in Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, pp. 155-164.

[Lukins'10] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H., (2010), "Bug localization using Latent Dirichlet Allocation", *Information and Software Technology*, vol. 52, no. 9, pp. 972-990.

[Lukoit'00] Lukoit, K., Wilde, N., Stowell, S., and Hennessey, T., (2000), "TraceGraph: Immediate Visual Location of Software Features ", in Proceedings of 16th IEEE International Conference on Software Maintenance (ICSM'00), Washington DC, USA, pp. 33-39.

[Marcus'03] Marcus, A., Feng, L., and Maletic, J. I., (2003), "3D Representations for Software Visualization", in Proceedings of 1st ACM Symposium on Software Visualization (SoftVis'03), San Diego, CA, June 11-13, pp. 27-36.

[Marcus'05a] Marcus, A. and Poshyvanyk, D., (2005a), "The Conceptual Cohesion of Classes", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30, pp. 133-142.

[Marcus'05b] Marcus, A. and Rajlich, V., (2005b), "Panel Summary: Identifications of Concepts, Features, and Concerns in Source Code", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30, pp. 718.

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

[Marcus'04] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., (2004), "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, November 9-12, pp. 214-223.

[Olszak'10] Olszak, A. and Jørgensen, B. N., (2010), "Featureous: A Tool for Feature-Centric Analysis of Java Software", in Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2, pp. 44-45.

[Petrenko'07] Petrenko, M., Poshyvanyk, D., Rajlich, V., and Buchta, J., (2007), "Teaching Software Evolution in Open Source", *IEEE Computer*, vol. 40, no. 11, pp. 25-31.

[Petrenko'08] Petrenko, M., Rajlich, V., and Vanciu, R., (2008), "Partial Domain Comprehension in Software Evolution and Maintenance", in *International Conference on Program Comprehension*.

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

[Poshyvanyk'07a] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., (2007a), "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, vol. 33, no. 6, June, pp. 420-432.

[Poshyvanyk'06b] Poshyvanyk, D., Marcus, A., and Dong, Y., (2006b), "JIRiSS - an Eclipse plug-in for Source Code Exploration", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, June 14-17, pp. 252-255.

[Poshyvanyk'05] Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A., (2005), "IRiSS - A Source Code Exploration Tool", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30, pp. 69-72.

[Poshyvanyk'07b] Poshyvanyk, D. and Marcus, D., (2007b), "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, June, pp. 37-48.

[Poshyvanyk'06c] Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D., (2006c), "Source Code Exploration with Google ", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, PA, pp. 334 - 338.

[Rajlich'04] Rajlich, V. and Gosavi, P., (2004), "Incremental Change in Object-Oriented Programming", in *IEEE Software*, pp. 2-9.

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

[Ratanotayanon'10] Ratanotayanon, S., Choi, H. J., and Sim, S. E., (2010), "Using Transitive changesets to Support Feature Location", in Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), Antwerp, Belgium, Sept 20-24, pp. 341-344.

[Ratiu'06] Ratiu, D. and Deissenboeck, F., (2006), "How Programs Represent Reality (and How They Don't)", in Proceedings of 13th Working Conference on Reverse Engineering (WCRE'06), pp. 83 - 92.

[Ratiu'07] Ratiu, D. and Deissenboeck, F., (2007), "From Reality to Programs and (Not Quite) Back Again", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Canada, pp. 91-102.

[Revelle'05] Revelle, M., Broadbent, T., and Coppit, D., (2005), "Understanding Concerns in Software: Insights Gained from Two Case Studies", in Proceedings of 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, USA, May, pp. 23-32.

[Revelle'10] Revelle, M., Dit, B., and Poshyvanyk, D., (2010), "Using Data Fusion and Web Mining to Support Feature Location in Software", in Proceedings of 18th IEEE International Conference on Program Comprehension (ICPC'10), Braga, Portugal, June 30 - July 2, pp. 14-23.

[Revelle'09] Revelle, M. and Poshyvanyk, D., (2009), "An Exploratory Study on Assessing Feature Location Techniques", in Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, British Columbia, Canada, May 17-19, pp. 218-222.

[Robillard'05a] Robillard, M., (2005a), "Automatic Generation of Suggestions for Program Investigation", in Proceedings of Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, September, pp. 11 - 20

[Robillard'08] Robillard, M. P., (2008), "Topology Analysis of Software Dependencies", *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, August, pp. 1-36.

[Robillard'04] Robillard, M. P., Coelho, W., and Murphy, G. C., (2004), "How effective developers investigate source code: an exploratory study", *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 12, pp. 889- 903.

[Robillard'02] Robillard, M. P. and Murphy, G. C., (2002), "Concern Graphs: Finding and describing concerns using structural program dependencies", in Proceedings of International conference on software engineering, pp. 406–416.

[Robillard'03a] Robillard, M. P. and Murphy, G. C., (2003a), "Automatically Inferring Concern Code from Program Investigation Activities", in Proceedings of 18th International Conference on Automated Software Engineering (ASE'03), Linz, Austria, October, pp. 225-234.

[Robillard'03b] Robillard, M. P. and Murphy, G. C., (2003b), "FEAT a tool for locating, describing, and analyzing concerns in source code", in Proceedings of 25th International Conference on Software Engineering (ICSE03), Portland, OR, May 3-10, pp. 822-823.

[Robillard'07a] Robillard, M. P. and Murphy, G. C., (2007a), "Representing concerns in source code", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1.

[Robillard'07b] Robillard, M. P., Shepherd, D., Hill, E., Vijay-Shanker, K., and Pollock, L., (2007b), "An Empirical Study of the Concept Assignment Problem". Montreal, Quebec, Canada: McGill University.

[Robillard'05b] Robillard, M. P. and Weigand-Warr, F., (2005b), "ConcernMapper: Simple View-Based Separation of Scattered Concerns", in Proceedings of Eclipse Technology Exchange at OOPSLA (ETX'05), pp. 65-69.

[Rohatgi'07] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., (2007), "Feature Location Based on Impact Analysis", in Proceedings of International Conference on Software Engineering and Applications (SEA'07).

[Rohatgi'08] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., (2008), "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis", in Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands, pp. 236-241.

[Rohatgi'09] Rohatgi, A., Hamou-Lhadj, A., and Rilling, J., (2009), "An Approach for Solving the Feature Location Problem by Measuring the Component Modification Impact", *IET Software*, vol. 3, no. 4, August, pp. 292-311.

[Safyallah'06] Safyallah, H. and Sartipi, K., (2006), "Dynamic Analysis of Software Systems using Execution Pattern Mining", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, June 14-17, pp. 84-88.

[Sahner'86] Sahner, R. A. and Trivedi, K. S., (1986), ""SHARPE: Symbolic Hierarchical Automated Reliability And Performance Evaluator, introduction and guide for users,"". Durham, NC: Duke University.

[Salah'04] Salah, M. and Mancoridis, S., (2004), "A hierarchy of dynamic software views: from object-interactions to feature-interactions", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, September 11-14, pp. 72-81.

[Salah'06] Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M., (2006), "Scenario-driven dynamic analysis for comprehending large software systems", in Proceedings of 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR'06), March 22-24, pp. 71-80.

[Salton'86] Salton, G. and McGill, M., (1986), *Introduction to Modern Information Retrieval*, New York, NY, USA, McGraw-Hill.

[Sartipi'10] Sartipi, K. and Safyallah, H., (2010), "Dynamic Knowledge Extraction from Software Systems using Sequential Pattern Mining", *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 20, no. 6, September, pp. 761-782.

[Saul'07] Saul, M. Z., Filkov, V., Devanbu, P., and Bird, C., (2007), "Recommending Random Walks", in Proceedings of 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia, pp. 15-24.

[Savage'10a] Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D., (2010a), "TopicXP: Exploring Topics in Source Code using Latent Dirichlet Allocation", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timişoara, Romania, September 12-18, pp. 1-6.

[Savage'10b] Savage, T., Revelle, M., and Poshyvanyk, D., (2010b), "FLAT^3: Feature Location and Textual Tracing Tool", in Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, pp. 255-258.

[Schach'01] Schach, S., (2001), *Object-Oriented and Classical Software Engineering*, 5 ed., New York, McGraw-Hill Higher Education.

[Shao'09] Shao, P. and Smith, R. K., (2009), "Feature location by IR modules and call graph", in Proceedings of ACM Annual Southeast Regional Conference, Clemson, South Carolina.

[Shepherd'07] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K., (2007), "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns", in Proceedings of 6th International Conference on Aspect Oriented Software Development (AOSD'07), pp. 212-224.

[Shepherd'06] Shepherd, D., Pollock, L., and Vijay-Shanker, K., (2006), "Towards Supporting On-Demand Virtual Remodularization Using Program Graphs", in Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, pp. 3-14.

[Shonle'04] Shonle, M., Neddenriep, J., and Griswold, W., (2004), "AspectBrowser for Eclipse: a case-study in plug-in retargeting", in Proceedings of OOPSLA workshop on eclipse technology eXchange, pp. 78-82.

[Simmons'06] Simmons, S., Edwards, D., Wilde, N., Homan, J., and Groble, M., (2006), "Industrial tools for the feature location problem: an exploratory study", *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 6, pp. 457-474.

[Somerville'01] Somerville, I., (2001), *Software Engineering*, Sixth ed., New Work, Addison-Wesley.

[Trifu'08] Trifu, M., (2008), "Using Dataflow Information for Concern Identification in Object-Oriented Software Systems", in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'08), pp. 193-202.

[Trifu'09] Trifu, M., (2009), "Improving the Dataflow-Based Concern Identification Approach", in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'09), pp. 109-118.

[Van Geet'09] Van Geet, J. and Demeyer, S., (2009), "Feature Location in COBOL Mainframe Systems: an Experience Report", in Proceedings of International Conference on Software Maintenance (ICSM'09), pp. 361-370.

[Walkinshaw'07] Walkinshaw, N., Roper, M., and Wood, M., (2007), "Feature Location and Extraction using Landmarks and Barriers", in Proceedings of International Conference on Software Maintenance (ICSM'07), Paris, France, pp. 54-63.

[Weigand-Warr'08] Weigand-Warr, F. and Robillard, M. P., (2008), "Suade: Topology-Based Searches for Software Investigation", in Proceedings of International Conference on Software Engineering, May, pp. 780-783.

[Wilde'01] Wilde, N., Buckellew, M., Page, H., and Rajlich, V., (2001), "A Case Study of Feature Location in Unstructured Legacy Fortran Code", in Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'01), pp. 68-76.

[Wilde'03] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L., (2003), "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, vol. 65, no. 2, February 15, pp. 105-114.

[Wilde'96] Wilde, N. and Casey, C., (1996), "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension", in Proceedings of International Conference on Software Maintenance (ICSM'96), pp. 270.

[Wilde'92] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., (1992), "Locating User Functionality in Old Code", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'92), Orlando, FL, November, pp. 200-205.

[Wilde'95] Wilde, N. and Scully, M., (1995), "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49-62.

[Wilde'02] Wilde, N., Simmons, S., Edwards, D., and Pounds, L., (2002), "But Where Does it DO That? Locating Features in a Distributed Simulation", in Proceedings of Fall Simulation Interoperability Workshop, Orlando, Florida.

[Wilson'10] Wilson, L. A., (2010), "Using Ontology Fragments in Concept Location", in Proceedings of 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania, Sept 12-18, pp. 1-2.

[Wohlin'99] Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnel, B., and Wesslen, A., (1999), *Experimentation in Software Engineering: An Introduction*, Amsterdam, Kluwer Academic Press.

[Wong'99] Wong, W. E., Gokhale, S. S., Horgan, J. R., and Trivedi, K. S., (1999), "Locating program features using execution slices", in Proceedings of IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99), March 24-27, pp. 194-203.

[Würsch'10] Würsch, M., Ghezzi, G., Reif, G., and Gall, H. C., (2010), "Supporting developers with natural language queries", in Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, May 2-8, pp. 165-174.

[Xie'06] Xie, X., Poshyvanyk, D., and Marcus, A., (2006), "3D Visualization for Concept Location in Source Code", in Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE'06), Shanghai, China, May 20-28, pp. 839-842.

[Zhai'04] Zhai, C. and Lafferty, J., (2004), "A Study of Smoothing Methods for Language Models Applied to Information Retrieval", *ACM Transactions on Information Systems*, vol. 22, no. 2, Apr. 2004, pp. 179-214.

[Zhao'04] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., (2004), "SNIAFL: Towards a Static Non-Interactive Approach to Feature Location", in Proceedings of 26th International Conference on Software Engineering (ICSE'04), May, pp. 293-303.

[Zhao'06] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F., (2006), "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.

Table 3 Classification of approaches and techniques from Section 4.2 to Section 4.9 within our taxonomy (refer to our online appendix for references and the list of systems evaluated)

Approach	Input			Derivative						Output						Prog. Lang.			Evaluation							
	Query	Scenario	Prog. Elem.	SC compilable	SC non-comp.	Dep. graph	Exec. trace	Historical inf.	Other	File/class	Method	Statement	Non-SC artifact	Ranked	Visualization	Java	C/C++	Other	Preliminary	Benchmark	Academic	Professional	Quantitative	Qualitative	Comparison	Unknown/none
Dynamic																										
[Wilde'95]		●		●			●		●		●						●		●			●	●	●		
[Wilde'92]		●		●			●		●		●					●	●		●					●		
[Eisenberg'05]		●		●			●		●		●			●	●	●			●				●	●	●	
[Wong'99]		●		●			●					●			●		●		●				●	●		
[Eisenbarth'01b]		●		●			●			●	●	●		●	●		●		●					●		
[Eisenbarth'01c]		●		●			●			●	●	●		●	●		●		●					●		
[Safyallah'06]		●		●			●		●		●						●		●					●		
[Sartipi'10]		●		●			●		●		●				●		●		●					●		
[Edwards'06]		●		●			●		●		●			●			●	●	●					●		
[Bohnet'08b]		●		●			●		●		●		●		●		●		●		●			●	●	
Static																										
[Chen'00]			●	●		●					●						●		●		●			●		
[Robillard'02]			●	●		●			●	●	●					●			●		●	●	●	●		
[Robillard'07a]			●	●		●			●	●	●					●			●		●			●		
[Robillard'05a]			●	●		●					●	●		●		●			●				●	●		
[Robillard'08]			●	●		●					●			●		●			●		●		●	●		
[Saul'07]			●	●		●					●			●			●		●				●	●	●	
[Trifu'08]			●			●					●					●			●					●		
[Trifu'09]			●			●					●					●			●				●	●	●	
Textual																										
[Petrenko'08]	●				●				●		●					●	●		●		●			●		
[Wilson'10]	●				●				●		●					●	●		●		●		●	●		
[Marcus'04]	●				●						●			●			●		●		●		●	●	●	
[Poshyvanyk'07b]	●				●				●		●			●	●	●			●		●		●	●		
[Cleary'07]	●				●				●		●			●		●			●				●		●	
[Cleary'09]	●				●				●		●			●		●			●				●	●	●	
[Gay'09]	●				●				●		●			●		●			●		●			●	●	
[Grant'08]					●				●		●			●			●		●					●		
[Shepherd'06]	●				●				●	●	●					●			●		●			●		
[Hill'09]	●				●				●		●			●		●				●		●	●	●	●	
[Abebe'10]	●				●				●	●				●			●		●		●		●	●		
[Würsch'10]	●				●				●		●					●			●							
Dynamic + Static																										
[Eisenbarth'01a]		●		●		●	●				●			●	●		●		●					●		
[Eisenbarth'03]		●		●		●	●																			●
[Koschke'05]		●		●			●		●		●	●		●	●		●		●				●	●		
[Antoniol'05]		●		●			●		●	●	●			●	●		●		●				●	●	●	
[Antoniol'06]		●		●			●		●	●	●			●	●	●	●		●				●	●	●	
[Rohatgi'07]		●		●		●	●			●				●		●			●					●		
[Rohatgi'08]		●		●		●	●			●				●		●			●					●		
[Rohatgi'09]		●		●		●	●			●				●		●			●					●		
[Walkinshaw'07]		●		●		●	●		●		●				●	●			●				●	●		
Dynamic + Textual																										
[Poshyvanyk'06a]	●	●		●	●		●				●			●			●		●				●	●	●	
[Poshyvanyk'07a]	●	●		●	●		●				●			●		●	●		●				●	●	●	
[Liu'07]	●	●		●	●		●				●			●		●			●		●		●	●	●	
[Asadi'10]		●		●	●		●				●					●			●				●	●		
[Revelle'10]	●	●		●	●	●	●		●		●			●		●				●			●	●	●	
[Hayashi'10a]	●			●	●		●		●		●			●		●			●				●	●		

Table 4 Classification of approaches and techniques from Section 4.2 to Section 4.9 within our taxonomy (continued)

Approach	Input			Derivative						Output						Prog. Lang.			Evaluation							
	Query	Scenario	Prog. Elem.	SC compilable	SC non-comp.	Dep. graph	Exec. trace	Historical inf.	Other	File/class	Method	Statement	Non-SC artifact	Ranked	Visualization	Java	C/C++	Other	Preliminary	Benchmark	Academic	Professional	Quantitative	Qualitative	Comparison	Unknown/none
Static + Textual																										
[Zhao'04]	●	.	●	●	●	●	.	.	●	.	●	.	●	.	.	.	●	.	●	.	.	.	●	●	●	.
[Zhao'06]	●	.	.	●	●	●	.	.	●	.	●	.	●	.	.	.	●	.	●	.	.	.	●	●	●	.
[Hill'07]	●	.	●	●	●	●	.	.	.	.	●	.	.	●	●	●	.	.	●	.	.	.	●	.	●	.
[Shao'09]	●	.	.	●	●	●	.	.	.	.	●	.	.	●	.	.	●	.	●	.	.	.	.	.	.	.
[Ratiu'06]	.	.	.	●	.	.	.	.	●	.	.	●	●	.	.	●	.	.	●	.	.	.	.	●	.	.
[Ratiu'07]	.	.	●	●	●	●	.	.	●	.	●	●	●	.	.	●	.	.	●	.	.	.	●	●	.	.
[Shepherd'07]	●	.	.	●	●	●	.	.	●	.	●	.	.	.	●	●	.	.	●	.	●	●	●	●	●	.
[Hayashi'10b]	●	.	.	●	.	.	.	.	●	.	●	.	.	●	●	●	.	.	●	.	.	.	●	●	.	.
Dynamic + Static + Textual																										
[Eaddy'08a]	●	.	.	●	●	●	●	.	●	.	●	.	.	●	.	●	.	.	.	●	.	.	●	●	●	.
Other																										
[Chen'01a]	●	.	.	.	●	.	.	●	●	●	.	●	.	.	●	.	●	.	●	.	●	.	.	●	●	.
[Cubranic'03]	●	.	●	.	●	.	.	●	●	●	●	.	●	●	.	●	.	.	●	.	●	.	.	●	●	.
[Cubranic'05]	●	.	●	.	●	.	.	●	●	●	●	.	●	●	.	●	.	.	●	.	●	●	●	●	.	.
[Cubranic'04]	●	.	●	.	●	.	.	●	●	●	●	.	●	●	.	●	.	.	●	.	●	●	.	●	●	.
[Robillard'03a]	.	.	.	●	.	.	.	.	●	.	●	●	.	.	.	●	.	.	●	.	●	.	.	●	.	.
[Ratanotayanon'10]	●	.	●	●	.	●	.	●	●	●	●	.	.	●	.	●	.	.	.	.	.	.	.	.	.	●

Table 5 Classification of tools and case studies

Approach	Input			Derivative						Output						Prog. Lang.			Evaluation							
	Query	Scenario	Prog. Elem.	SC compilable	SC non-comp.	Dep. graph	Exec. trace	Historical inf.	Other	File/class	Method	Statement	Non-SC artifact	Ranked	Visualization	Java	C/C++	Other	Preliminary	Benchmark	Academic	Professional	Quantitative	Qualitative	Comparison	Unknown/none
Tools for Dynamic FL																										
[Lukoit'00]		●		●			●		●		●	●			●		●		●					●		
[Egyed'07]		●		●			●				●				●	●	●		●							
[Olszak'10]		●		●			●		●		●				●	●										●
Tools for Static FL																										
[Chen'01b]			●	●		●					●				●		●		●		●		●	●	●	
[Buckner'05]						●										●								●		●
[Weigand-Warr'08]			●	●		●					●			●		●			●					●		
Tools for Textual FL																										
[Poshyvanyk'06c]	●				●					●				●		●			●		●		●	●	●	
[Poshyvanyk'05]	●				●						●						●									●
[Poshyvanyk'06b]	●				●						●					●										●
[Xie'06]	●				●						●				●		●									●
[Cleary'06]				●					●	●	●			●		●										●
Other Tools for FL																										
[Robillard'03b]			●	●		●			●	●	●					●										●
[Robillard'05b]	●				●				●		●					●										●
[Savage'10b]	●	●		●	●		●				●			●	●	●										●
[Savage'10a]	●			●		●				●	●			●	●	●			●		●		●	●	●	
[Bohnet'06b]		●		●		●	●		●		●				●		●							●		
[Bohnet'06a]		●		●		●	●		●	●					●		●									●
[Bohnet'08a]		●	●	●		●	●				●				●		●									●
[Bohnet'07b]																	●									
[Bohnet'07a]		●		●							●				●		●							●		●
Case Studies																										
[Wilde'96]		●		●			●		●		●			●			●		●					●		
[Wilde'01]	●	●	●	●	●	●	●		●		●			●	●			●	●		●	●		●	●	
[Wilde'03]		●	●	●		●	●		●		●			●	●			●	●		●	●		●	●	
[Ibrahim'03]		●		●			●		●		●			●			●		●					●		
[Marcus'05c]	●		●	●	●	●					●	●				●	●		●		●		●	●		
[Revelle'09]	●	●	●	●	●	●	●				●			●		●			●		●			●	●	
[Simmons'06]		●		●		●	●				●				●		●		●		●		●	●		
[Van Geet'09]		●		●			●		●		●					●		●	●					●		
[Revelle'05]			●		●				●			●				●	●		●		●		●		●	