A Survey of Feature Location Techniques

Julia Rubin and Marsha Chechik

Abstract Feature location techniques aim at locating software artifacts that implement a specific program functionality, a.k.a. *a feature*. These techniques support developers during various activities such as software maintenance, aspect- or feature-oriented refactoring, and others. For example, detecting artifacts that correspond to product line features can assist the transition from unstructured to systematic reuse approaches promoted by software product line engineering (SPLE). Managing features, as well as the traceability between these features and the artifacts that implement them, is an essential task of the SPLE domain engineering phase, during which the product line resources are specified, designed, and implemented. In this chapter, we provide an overview of existing feature location techniques. We describe their implementation strategies and exemplify the techniques on a realistic use-case. We also discuss their properties, strengths, and weaknesses and provide guidelines that can be used by practitioners when deciding which feature location technique to choose. Our survey shows that none of the existing feature location techniques are designed to consider families of related products and only treat different products of a product line as individual, unrelated entities. We thus discuss possible directions for leveraging SPLE architectures in order to improve the feature location process.

Keywords Feature location • Software maintenance • Software product lines

---

J. Rubin (✉)
IBM Research, Haifa, Israel and University of Toronto, Canada
e-mail: mjulia@il.ibm.com

M. Chechik
University of Toronto, Canada
e-mail: chechik@cs.toronto.edu

I. Reinhartz-Berger et al. (eds.), *Domain Engineering*,

DOI 10.1007/978-3-642-36654-3_2, © Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Software product line engineering (SPLE) techniques [10, 25] capitalize on identifying and managing *common* and *variable* product line features across a product portfolio. SPLE promotes *systematic* software reuse by leveraging the knowledge about the set of available features, relationships among the features and relationships between the features and software artifacts that implement them. However, in reality, software families—collections of related software products—often emerge ad hoc, from experiences in successfully addressed markets with similar, yet not identical needs. Since it is difficult to foresee these needs a priori and hence to design a software product line upfront, software developers often create new products by using one or more of the available technology-driven software reuse techniques such as duplication (the "clone-and-own" approach), source control branching, preprocessor directives, and more.

Essential steps for unfolding the complexity of existing implementations and assisting their transformation to systematic SPLE reuse approaches include identification of implemented features and detection of software artifacts that realize those features. While the set of available features in many cases is specified by the product documentation and reports, the relationship between the features and their corresponding implementation is rarely documented. Identification of such relationships is the main goal of feature location techniques.

Rajlich and Chen [8] represent a feature (a.k.a. a *concept*) as a triple consisting of a *name*, *intension*, and *extension*. The name is the label that identifies the feature; intension explains the meaning of the feature; and extension is a set of artifacts that realize the feature. *Location: intension* $\rightarrow$ *extension* is identified by the authors as one of the six fundamental program comprehension processes. Its application to features is the subject of this survey.

In the remainder of the chapter, we illustrate the surveyed concepts using a problem of locating the *automatic save file* feature, previously studied in [32], in the code of the Freemind[1] open source mind-mapping tool. A snippet of Freemind's call graph is shown in Fig. 1. Shaded elements in the graph contribute to the implementation of the *automatic save file* feature—they are the feature *extension* which we want to locate. Feature *intension* can be given, for example, by the natural language query "*automatic save file*," describing the feature.[2]

The feature is mainly implemented by two methods of the `MindMapMapModel` subclass `doAutomaticSave`: the constructor and the method `run` (elements #1 and #2). `doAutomaticSave` class is initiated by the `MindMapMapModel`'s constructor (element #4), as shown in Fig. 2. The constructor assigns values to several configuration parameters related to the *automatic save file* function and then registers the `doAutomaticSave` class on the scheduling queue. This initiates the

[1] http://freemind.sourceforge.net.

[2] We denote features by *italic font*, place natural language queries "*in quotes*," and denote code elements by a `monospaced font`.

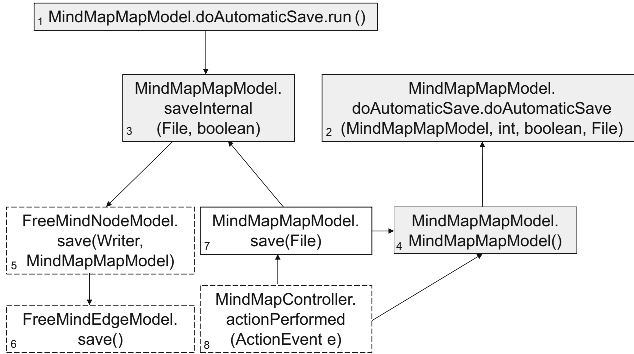


Fig. 1 The *automatic save file* call graph snippet

```
public MindMapMapModel( MindMapNodeModel root,
    ⋮                   FreeMindMain frame ) {
  // automatic save:
  timerForAutomaticSaving = new Timer();
  int delay = Integer.parseInt(getFrame().
    getProperty("time_for_automatic_save"));
  int numberOfTempFiles = Integer.parseInt(getFrame().
    getProperty("number_of_different_files_for_automatic_save"));
  boolean filesShouldBeDeletedAfterShutdown = Tools.
    safeEquals(getFrame().
    getProperty("delete_automatic_save_at_exit"),"true");
  String path = getFrame().getProperty("path_to_automatic_saves");
    ⋮
  timerForAutomaticSaving.schedule(new doAutomaticSave(
    this, numberOfTempFiles,
    filesShouldBeDeletedAfterShutdown, dirToStore),
    delay, delay);
  );
}
```

Fig. 2 The `MindMapMapModel` code snippet

class's `run` method (element #1) which subsequently calls the `saveInternal` method (element #3) responsible for performing the *save* operation.

Obviously, not all program methods contribute to the *automatic save file* feature. For example, element #3 also initiates a call to `FreeMindNodeModel`'s `save( Writer, MindMapMapModel)` method (element #5), which, in turn, calls element #6–`save(Writer, MindMapMapModel)`. Both of these methods are irrelevant to the specifics of the *automatic save file* implementation. Element #3 itself is called by element #7 (`MindMapMapMode`'s `save(File)` method), which is called by element #8 (`MindMapController`'s `actionPerformed(ActionEvent)`). These methods are also not relevant to the feature implementation because they handle a *user-triggered save* operation instead of *automatic save*. In fact, element #8 initiates calls to an additional 24 methods, all of which are irrelevant to the implementation of the feature. In Fig. 1, irrelevant methods are not shaded.

While all feature location approaches share the same goal—establishing traceability between a specific feature of interest specified by the user (feature intension) and the artifacts implementing that feature (feature extension), they differ substantially in the underlying design principles, as well as in assumptions they make on their input (representation of the intension). In this chapter, we provide an in-depth description of 24 existing feature location techniques and their underlying technologies. We exemplify them on a small but realistic program snippet of the Freemind software introduced above and discuss criteria for choosing a feature location technique based on the qualities of the input program. We also assess the techniques by the amount of required user interaction.

Our specific interest is in applying feature location techniques in the context of software families where a feature can be implemented by multiple products. However, none of the existing techniques explicitly consider collections of related products when performing feature location: the techniques are rather applied to these products as if these are unrelated, singular entities. Thus, another contribution of our work is a discussion of research directions towards a more efficient feature location, taking advantage of existing families of related products (see Sect. 6).

A systematic literature survey of 89 articles related to feature location is available in [11]. That survey provides a broad overview of existing feature definition and location techniques, techniques for feature representation and visualization, available tools and performed user studies. The purpose of that work is organizing, classifying and structuring existing work in the field and discussing open issues and future directions. Even though 22 out of the 24 techniques surveyed here are covered by [11], our work has a complementary nature. We focus only on *automated feature location* techniques while providing insights about the implementation details, exemplifying the approaches and discussing how to select one in real-life settings. The intended audience of our survey is practitioners aiming to apply a feature location technique for establishing traceability between the features of their products and the implementation of these features. As such, these practitioners have to understand the implementation details and properties of the available approaches in order to choose one that fits their needs.

The rest of the chapter is organized as follows. In Sect. 2, we start by introducing basic technologies used by several feature location techniques. Section 3 introduces the classification that we use for the surveyed feature location techniques. A detailed description of the techniques themselves is provided in Sect. 4. We discuss criteria used when selecting a feature location technique in Sect. 5. Section 6 concludes our survey and presents directions for possible future work on feature location in the context of SPLE.

2 Basic Underlying Technologies

In this section, we introduce basic technologies commonly used by feature location techniques, describe each technology, and demonstrate it on the example in Sect. 1.

Objects/Attributes	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
`action`								√
`automatic`	√	√						
`controller`								√
`do`	√	√						
`file`								
`free`					√	√		
`internal`			√					
`map`	√	√	√	√			√	√
`mind`	√	√	√	√	√	√	√	√
`model`	√	√	√	√	√	√	√	
`node`					√	√		
`performed`								√
`run`	√							
`save`	√	√	√		√	√	√	

Fig. 3 Formal context for the example in Fig. 1. Objects are method names, attributes are tokens of the names

2.1 Formal Concept Analysis

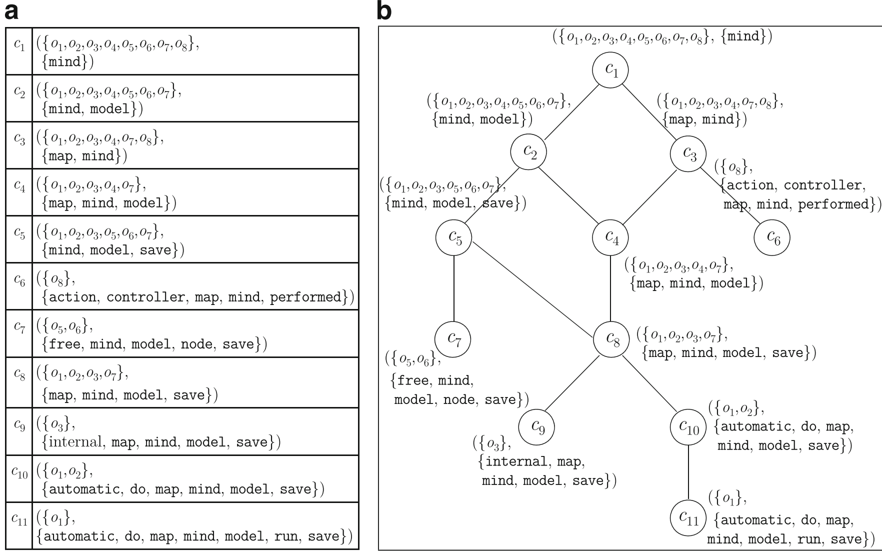
Formal Concept Analysis (FCA) [16] is a branch of mathematical lattice theory that provides means to identify meaningful *groupings* of objects that share common attributes. Groupings are identified by analyzing binary relations between the set of all objects and all attributes. FCA also provides a theoretical model to analyze hierarchies of these identified groupings.

The main goal of FCA is to define a *concept* as a unit of two parts: *extension* and *intension*.[3] The extension of a concept covers all the objects that belong to the concept, while the intension comprises all the attributes, which are shared by all the objects under consideration. In order to apply FCA, the *formal context* of objects and their respective attributes is necessary. The formal context is an incidence table indicating which attributes are possessed by each object. An example of such a table is shown in Fig. 3, where objects are names of methods in Fig. 1 and attributes are individual words obtained by tokenizing and lowercasing these names. For example, object o_1 corresponds to element #1 in Fig. 1 and is tokenized to attributes `automatic`, `do`, `map`, `mind`, `model`, `run`, `save`, which are "checked" in Fig. 3.

From the formal context, FCA generates a set of concepts where every concept is a maximal collection of objects that possess common attributes. Figure 3a shows all concepts generated for the formal context in Fig. 3.

Formally, given a set of objects O, a set of attributes A, and a binary relationship between objects and attributes R, the *set of common attributes* is defined as $\sigma(O) = \{a \in A \mid (o, a) \in R \ \forall o \in O\}$. Analogously, *the set of common objects* is defined as

[3]These are not to be confused with the extension and intension of a feature.



c_1	({$o_1,o_2,o_3,o_4,o_5,o_6,o_7,o_8$}, {mind})
c_2	({$o_1,o_2,o_3,o_4,o_5,o_6,o_7$}, {mind, model})
c_3	({o_1,o_2,o_3,o_4,o_7,o_8}, {map, mind})
c_4	({o_1,o_2,o_3,o_4,o_7}, {map, mind, model})
c_5	({o_1,o_2,o_3,o_5,o_6,o_7}, {mind, model, save})
c_6	({o_8}, {action, controller, map, mind, performed})
c_7	({o_5,o_6}, {free, mind, model, node, save})
c_8	({o_1,o_2,o_3,o_7}, {map, mind, model, save})
c_9	({o_3}, {internal, map, mind, model, save})
c_{10}	({o_1,o_2}, {automatic, do, map, mind, model, save})
c_{11}	({o_1}, {automatic, do, map, mind, model, run, save})

Fig. 4 Concepts and the corresponding concept lattice generated for the formal context in Fig. 3. (**a**) Concept, (**b**) Concept lattice

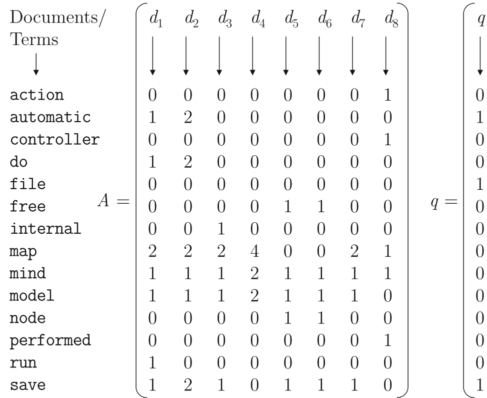
$\rho(O) = \{o \in O \mid (o,a) \in R\ \forall a \in A\}$. For example, for the relationship R encoded in Fig. 3, $\sigma(o_4) = \{\texttt{map}, \texttt{mind}, \texttt{model}\}$ and $\rho(\texttt{automatic}, \texttt{do}) = \{o_1, o_2\}$.

A *concept* is a pair of sets (O, A) such that $A = \rho(O)$ and $O = \sigma(A)$. O is considered to be the *extension* of the concept and A is the *intension* of the concept. The set of all concepts of a given formal context forms a partial order via the superconcept-subconcept ordering $\leq$: $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$, or, dually, $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_2 \subseteq A_1$.

The set of all concepts of a given formal context and the partial order $\leq$ form a *concept lattice*, as shown in Fig. 4b. In our example, this lattice represents a taxonomy of tokens used for naming the methods—from the most generic used by all methods (the root element c_1, which represents the term `mind` used in all names) to the more specific names depicted as leaves (e.g., c_6 which represents unique terms `action`, `controller` and `performed` used in the name of element #8).

2.2 *Latent Semantic Indexing*

Latent semantic indexing (LSI) [21] is an automatic mathematical/statistical technique that analyzes the relationships between queries and passages in large bodies of text. It constructs vector representations of both a user query and a corpus of text documents by encoding them as a *term-by-document co-occurrence matrix*. Each row in the matrix stands for a unique word, and each column stands for a text passage or a query. Each cell contains the frequency with which the word of its row appears in the passage denoted by its column.



$A =$ (Documents/Terms), $q =$

Documents/Terms	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	q
action	0	0	0	0	0	0	0	1	0
automatic	1	2	0	0	0	0	0	0	1
controller	0	0	0	0	0	0	0	1	0
do	1	2	0	0	0	0	0	0	0
file	0	0	0	0	0	0	0	0	1
free	0	0	0	0	1	1	0	0	0
internal	0	0	1	0	0	0	0	0	0
map	2	2	2	4	0	0	2	1	0
mind	1	1	1	2	1	1	1	1	0
model	1	1	1	2	1	1	1	0	0
node	0	0	0	0	1	1	0	0	0
performed	0	0	0	0	0	0	0	1	0
run	1	0	0	0	0	0	0	0	0
save	1	2	1	0	1	1	1	0	1

Fig. 5 Term-by-document co-occurrence matrix for the example in Fig. 1. Documents are method names, terms are tokens of the names and the query

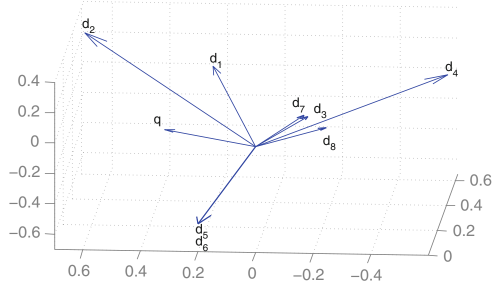


Fig. 6 Vectorial representation of the documents and the query in Fig. 5

Figure 5 shows such an encoding for the example in Fig. 1, where "documents" are method names, the query "*automatic save file*" is given by the user, and the set of all terms is obtained by tokenizing, lowercasing, and alphabetically ordering strings of both the documents and the query. In Fig. 5, matrix A represents the encoding of the documents and matrix q represents the encoding of the query. Vector representations of the documents and the query are obtained by normalizing and decomposing the *term-by-document co-occurrence matrix* using a matrix factorization technique called *singular value decomposition* [21]. Figure 6 shows the vector representation of the documents $d_1 \ldots d_8$ and the query q in Fig. 5 in a three-dimensional space.

The similarity between a document and a query is typically measured by the cosine between their corresponding vectors. The similarity increases as the vectors point "in the same general direction," i.e., as more terms are shared between the documents. For the example in Fig. 6, document d_2 is the most similar to the query, while d_8 is the least similar. The exact similarity measures between the document and the query, as calculated by LSI, are summarized in Table 1. It is common to consider documents with positive similarity values as related to the query of interest (i.e., d_1, d_2, d_5 and d_6 in our example), while those with negative similarity values (i.e., d_3, d_4, d_7 and d_8)—as unrelated.

Table 1 Similarity of the documents and the query in Fig. 5 as calculated by LSI

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
0.6319	0.8897	−0.2034	−0.5491	0.2099	0.2099	−0.1739	−0.6852

2.3 Term Frequency: Inverse Document Frequency Metric

Term frequency—inverse document frequency (*tf-idf*) is a statistical measure often used by IR techniques to evaluate how important a term is to a specific document in the context of a set of documents (*corpus*). Intuitively, the more frequently a term occurs in the document, the more relevant the document is to the term. That is, the relevance of a specific document d to a term t is measured by *document frequency* ($tf(t,d)$). For the example in Fig. 5 where "documents" are names of methods in Fig. 1, the term `save` appears twice in d_2, thus $tf(\texttt{save}, d_2) = 2$.

The drawback of term frequency is that uninformative terms appearing throughout the set D of all documents can distract from less frequent, but relevant, terms. Intuitively, the more documents include a term, the less this term discriminates between documents. The *inverse document frequency*, *idf(t)*, is then calculated as follows: $idf(t) = \log(\frac{|D|}{|\{d \in D \mid t \in d\}|})$. The *tf-idf* score of a term w.r.t. a document is calculated by multiplying its *tf* and *idf* scores: *tf-idf(t,d)* $= tf(t,d) \times idf(t)$. In our example, $idf(\texttt{save}) = \log(\frac{8}{6}) = 0.12$ and $tf\text{-}idf(\texttt{save}, d_2) = 2 \times 0.12 = 0.24$.

Given a query which contains multiple terms, the *tf-idf* score of a document with respect to the query is commonly calculated by adding *tf-idf* scores of all query terms. For example, the *tf-idf* score of d_2 with respect to the query "*automatic save file*" is 1.44, while d_3 score with respect to the same query is 0.12.

2.4 Hyper-link Induced Topic Search

Hyper-link Induced Topic Search (HITS) is a page ranking algorithm for Web mining introduced by Kleinberg [19]. The algorithm considers two forms of web pages—*hubs* (pages which act as resource lists) and *authorities* (pages with important content). A good hub points to many authoritative pages whereas a good authority page is pointed to by many good hub pages.

The HITS algorithm operates on a directed graph, whose nodes represent pages and whose edges correspond to links. Authority and hub scores for each page p (denoted by A_p and H_p, respectively) are defined in terms of each other: $A_p = \sum_{\{q \mid q \text{ points to } p\}} H_q$ and $H_p = \sum_{\{q \mid p \text{ points to } q\}} A_q$. The algorithm initializes hub and authority scores of each page to 1 and performs a series of iterations. Each iteration calculates and normalizes the hub (authority) value of each page. It does so by dividing the value by the square root of the sum of squares of all hub (authority) values for the pages that it points to (pointed by). The algorithm stops when it reaches a fixpoint or a maximum number of iterations.

When applied to code, HITS scores methods in a program based on their "strength" as hubs—aggregators of functionality, i.e., methods that call many others, and authorities—those that implement some functionality without aggregation. For the example in Fig. 1, elements #2 and #6 are authorities as they do not call any other methods and thus their hub score is 0. Elements #1 and #8 are hubs as they are not called by other methods. Thus, their authority score is 0. Elements #3 and #4 get a higher authority score than other elements as they are called by two methods each, while elements #7 and #8 get a higher hub score than the rest as they call two methods each.

3 Classification and Methodology

In this section, we discuss the classification of feature location techniques that we use for organizing our survey. We also discuss main properties that we highlight for each technique.

Primarily, feature location techniques can be divided into *dynamic* which collect information about a program at runtime and *static* which do not involve program execution. The techniques also differ in the way they assist the user in the process of interpreting the produced results. Some only present an (unsorted) list of artifacts considered relevant to the feature of interest; we refer to these as *plain output* techniques. Others provide additional information about the output elements, such as their relative ranking based on the perceived relevance to the feature of interest or automated and guided output exploration process which suggests the order and the number of output elements to consider; we refer to these as *guided output* techniques. Figure 7 presents the surveyed techniques, dependencies between them and their primary categorization.

Feature location approaches can rely on *program dependence analysis (PDA)* that leverages static dependencies between program elements; *information retrieval (IR)* techniques—in particular, *LSI*, *tf-idf* and others, that leverage information embedded in program identifier names and comments; *change set analysis* that leverages historical information and more. While dynamic approaches collect precise information about the program execution, they are safe only with respect to the input that was actually considered during runtime to gather the information, and generalizing from this data may not be safe [20]. In addition, while generally a *feature* is a realization of a system requirement—either functional or non-functional—executable test-cases or scenarios can exhibit only *functional* requirements of the system that are visible at the user level. Thus, dynamic feature location techniques can detect only functional features. On the other hand, static approaches can locate any type of feature and yield safe information, but because many interesting properties of programs are statically undecidable in general, static analysis is bound to approximate solutions which may be too imprecise in practice. Dynamic analysis yields "under-approximation" and thus might suffer

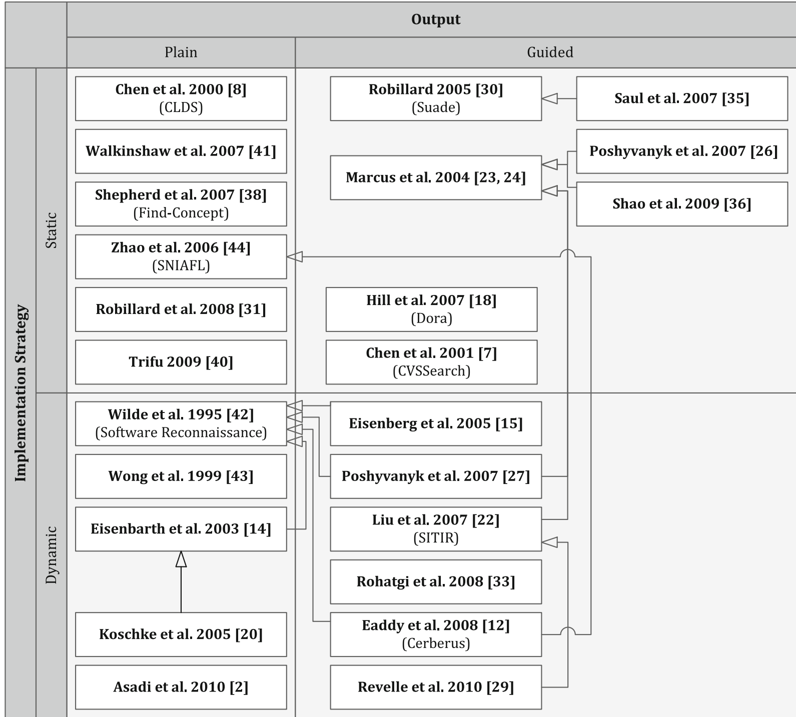


Fig. 7 Surveyed techniques and their categorization

from many false-negative results while static analysis yields "over-approximation" and thus might have many false-positives. In order to find a middle ground, *hybrid* approaches combine several techniques.

Based on the chosen implementation technique, the analyzed program can be represented as a *program dependence graph (PDG)*, a set of text documents representing software elements, an instrumented executable that is used by dynamic techniques and more. Figure 8 provides detailed information about each of the surveyed techniques, listing its underlying technology, the chosen program representation, the type of user input, as well as the amount of required user interaction, ranging from low (denoted by "+") to high (denoted by "+ + +").

4 Feature Location Techniques

In this section, we describe automated feature location techniques from the literature. As discussed in Sect. 1, we focus on those techniques that assist the user with feature *location* rather than feature definition or visualization. Static approaches

		Technique	Underlying Technology	Program Representation	Input	User Interaction
Static	Plain	**Chen et al. [8]** (CLDS)	PDA	PDG	method or global variable	+++
		Walkinshaw at al. [41]	PDA	call graph	two sets of methods	+
		Shepherd et al. [38] (Find-concept)	PDA, NLP	AOIG	query	++
		Zhao et al. [44] (SNIAFL)	TF-IDF, vector space model, PDA	BRCG	set of queries	+
		Robillard et al. [31]	clustering algorithms	change history transactions	set of elements	+
		Trifu [40]	PDA	concern graph	set of variables	+++
	Guided	**Robillard [30]** (Suade)	PDA	PDG	set of methods and global variables	++
		Saul et al. [35]	PDA, web-mining algorithm	call graph	method	++
		Marcus et al. [23, 24]	LSI	text docs for software elements	query	+
		Poshyvanyk et al. [26]	LSI, FCA on retrieved docs	text docs for software elements	query	+
		Shao et al. [36]	LSI, PDA	call graph, text docs for software elements	query	+
		Hill et al. [18] (Dora)	PDA, TF-IDF	call graph, text docs for software elements	method, query	+
		Chen et al. [7] (CVSSearch)	textual search	lines of code, CVS comments	query	+
Dynamic	Plain	**Wilde et al. [42]** (Sw. Reconnaissance)	trace analysis	executable	set of test cases	+++
		Wong et al. [43]	trace analysis	executable	set of test cases	+++
		Eisenbarth et al. [14]	trace analysis (FCA), PDA	executable, PDG	set of scenarios	+++
		Koschke et al. [20]	trace analysis (FCA), PDA	executable, statement dependency graph	set of scenarios	+++
		Asadi et al. [2]	trace analysis, LSI, genetic optimization	executable, text docs for methods	set of scenarios	++
	Guided	**Eisenberg et al. [15]**	trace analysis	executable	set of test cases	++
		Poshyvanyk et al. [27]	trace analysis, LSI	executable, text docs for methods	set of scenarios, query	+++
		Liu et al. [22] (SITIR)	trace analysis, LSI	executable, text docs for methods	scenario, query	+
		Rohatgi et al. [33]	trace analysis, impact analysis	executable, class dependency graph	set of scenarios	++
		Eaddy et al. [12] (Cerberus)	PDA, trace analysis, TF-IDF, vector space model	PDG, executable, text docs for software elements	set of queries, set of scenarios	+++
		Revelle et al. [29]	trace analysis, LSI, web-mining algorithm	executable, text docs for methods	scenario, query	+

Fig. 8 Underlying technology, program representation, and input type of the surveyed techniques

(those that do no require program execution) are described in Sect. 4.1; dynamic are in Sect. 4.2.

4.1 *Static Feature Location Techniques*

In this section, we describe techniques that rely on static program analysis for locating features in the source code.

4.1.1 Plain Output

Chen et al. [8] present one of the earliest static computer-assisted feature location approaches based on *PDA*. The analyzed program is represented as a *PDG* whose nodes are methods or global variables and edges are method invocations or data access links (the paper refers to the PDG as the abstract system dependence graph). Given an initial element of interest—either a function or a global variable—the approach allows the user to explore the PDG interactively, node-by-node, while storing visited nodes in a *search graph*. The user decides whether the visited node is related to the feature and marks related nodes as such. The process stops when the user is satisfied with the set of found nodes, and outputs the set of relevant nodes aggregated in the search graph. For the example in Fig. 1, the system generates the depicted call graph from the source code and interactively guides the user through its explanation. The technique relies on extensive user interaction (denoted by "+++" in Fig. 8), and thus provides the user with "intelligent assistance" [6] rather than being a heuristic-based technique aiming to determine relevant program elements automatically.

Walkinshaw et al. [41] provide additional automation to the feature location process based on PDA. The analyzed program is represented as a *call graph*—a subgraph of PDG containing only methods and method invocations. As input, the system accepts two sets of methods: *landmark*—thought to be essential for the implementation of the feature, and *barrier*—thought to be irrelevant to the implementation. For the example in Fig. 1, landmark methods could be elements #1 and #2, while barrier methods—#5 and #7. The system computes a *hammock graph* which contains all of the direct paths between the landmarks. That is, a method call belongs to the hammock graphs only if it is on a direct path between a pair of landmark methods. Additional potentially relevant methods are added to the graph using intra-procedural *backward slicing* [39] (with the call sites that spawn calls in the hammock graph as slicing criteria). Since slicing tends to produce graphs that are too large for practical purposes, barrier methods are used to eliminate irrelevant sections of the graph: all incoming and outgoing call graph edges of barrier methods are removed, and thus these are not traversed during the slice computation. The approach outputs all elements of the resulting graph as relevant to the feature.

In our example in Fig. 1, no direct call paths exist between elements #1 and #2; thus, the approach is unable to find additional relevant elements under the given input. The technique is largely automated and does not require extensive user interaction (denoted by "+" in Fig. 8) other than providing and possibly refining the input sets of methods.

Shepherd et al. [38] attempt to locate action-oriented concepts in object-oriented programs using domain knowledge embedded in the source code through identifier names (methods and local variables) and comments. It relies on the assumption that verbs in object-oriented programs correspond to methods, whereas nouns correspond to objects.

The analyzed program is represented as an *action-oriented identifier graph model (AOIG)* [37] where the actions (i.e., verbs) are supplemented with direct

objects of each action (i.e., objects on which the verb acts). For example, the verb `save` in Fig. 1 can act on different objects in a single program, such as `MindMapMapModel` and `MindMapNodeModel`; these are the direct objects of `save`. An AOIG representation of a program contains four kinds of nodes: *verb nodes*, one for each distinct verb in the program; *direct object (DO) nodes*, one for each unique direct object in the program; *verb-DO nodes*, one for each verb-DO pair identified in the program (a verb or a direct object can be part of several verb-DO pairs); and *use nodes*, one for each occurrence of a verb-DO pair in comments or source code of the program. An AOIG has two kinds of edges: *pairing edges* connecting each verb or DO node to verb-DO pairs that use them, and *use edges* connecting each verb-DO pair to all of its use nodes.

As an input, the user formulates a query describing the feature of interest and *decomposes* it into a set of pairs (verb, direct object). The technique helps the user to refine the input query by collecting verbs and direct objects that are similar (i.e., different forms of words, and synonyms) to the input verbs and direct objects, respectively, as well as words collocated with those in the query, based on the verb-DO pairs of the program AOIG. For example, `MindMapMapModel` is collocated with `MindMapNodeModel` in verb-DO pairs for the verb `save`. The collected terms are ranked by their "closeness" to the words in the query based on the frequency of collocation with the words in the query and on configurable weight given to synonyms. Ten best-ranked terms are presented to the user. The system then recommends that the user augment the query with these terms as well as with program methods that match the current query.

Once the user is satisfied with the query, the system searches the AOIG for all verb-DO pairs that contain the words of the query. It extracts all methods where the found pairs are used and applies PDA to detect call relationships between the extracted methods. The system then generates the *result graph* in which nodes represent detected methods and edges represent identified structural relationships between them. The graph is returned to the user.

For our example in Fig. 1, the input query (`doAutomaticSave`, `MindMapMapModel`) might get expanded by the user with the terms `save` and `saveInternal`, because they are collocated with `MindMapMapModel`. Then, the system outputs elements #1 through #4 and #7, together with the corresponding call graph fragment. The technique requires a fair amount of user interaction to construct and refine the input query, and thus is marked with "++" in Fig. 8.

Zhao et al. [44] accept a set of feature descriptions as input and focus on locating the *specific* and the *relevant* functions of each feature using PDA and IR technologies. The specific functions of a feature are those definitely used to implement it but are not used by other features. The relevant functions of a feature are those involved in the implementation of the feature. Obviously, the specific function set is a subset of the relevant function set for every feature.

The analyzed program is represented as a *Branch-Reserving Call Graph (BRCG)* [28]—an expansion of the call graph with branching and sequential information, which is used to construct the pseudo execution traces for each feature. Each node in the BRCG is a function, a branch, or a return statement. Loops are regarded as

two branch statements: one going through the loop body and the other one exiting immediately. The nodes are related either sequentially, for statements executed one after another, or conditionally, for alternative outcomes of a branch.

The system receives a paragraph of text as a description of each feature. The text can be obtained from the requirements documentation or be provided by a domain expert. It transforms each feature description into a set of index terms (considering only nouns and verbs and using their normalized form). These will be used as documents. The system then extracts the names of each method and its parameters, separating identifiers using known coding styles (e.g., using the underline "_" to separate words) and transforms them into index terms. These will be used as queries.

To reveal the connections between features and functions, documents (feature descriptions) are ranked for each query (function) using the vector space models [3, pp. 27–30]—a technique which, similar to LSI, treats queries and documents as vectors constructed by the index terms. Unlike LSI, the weights of index term in documents and queries are calculated using the *tf-idf* metric (see Sect. 2.3) between the term and the document or query, respectively. For the example in Fig. 1, *automatic save file* could be a document while "*mind map model do automatic save*" could be a query corresponding to the element #2. For the vector space model, the weight of the term `save` in the query is 0.24, as calculated in Sect. 2.3. Note that LSI calculates this weight as being 2 (see the value of the term `save` in the column that corresponds to d_2 in Fig. 5).

Similarity between a document and a query is computed as a cosine of the angle between their corresponding vectors, as for LSI. For each document (feature), the system creates a sorted list of queries (functions), ranked by their similarity degrees and identifies a pair of functions with the largest difference between scores. All functions before this pair, called *a division point*, are considered *initial specific functions* to the feature. In our example, these are elements #1 and #2.

Next, the system analyzes the program's BRCG and filters out all branches that do not contain any of the initial specific functions of the feature, because those are likely not relevant; all remaining functions are marked as *relevant*. Functions relevant to exactly one feature are marked as *specific* to that feature.

The system also builds pseudo-execution traces for each feature by traversing the pruned BRCG and returns those to the user. For our example in Fig. 1, BRCG is rooted in element #8. Since there is no direct call to element #1 (the call is performed via an event queue—see the last statement in Fig. 2), the technique returns only those branches that contain element #2, that is, elements #8, #7, and #4. The technique requires no user interaction besides the definition and the refinement of the input feature descriptions, as reflected by "+" in Fig. 8.

Robillard et al. [31] propose searching the change history (*change transactions*) of a software system to identify clusters of program elements related to a task. The analyzed program is represented as a set of program elements such as fields and methods, as well as change history transactions that capture modifications of these elements. The system considers all available transactions and filters out those with more than 20 or fewer than four elements. The thresholds are set empirically: experiments revealed that large transactions generate overly large clusters that

would require developers to spend an unreasonable amount of effort to study, while small transactions cannot be clustered efficiently. The system then clusters the remaining transactions based on the number of overlapping elements using a standard clustering algorithm.

Next, given a small set of elements related to a feature of interest (usually two or three), the system extracts clusters containing all input elements and removes those satisfying the following conditions:

1. An input element appears in at least 3% of the transactions of the cluster. The rationale is that querying the change history for elements that are being continuously modified (and thus are central or critical elements to the entire system) returns too many recommendations to be useful.
2. The degree of overlap between elements that correspond to the transactions in a cluster is lower than 0.6. The rationale is that these clusters do not represent changes that are associated with a high-level concept.
3. The number of transactions in a cluster is less than 3. The rationale is to avoid returning results that are single transactions or very small groups of transactions which may have been spontaneously clustered. However, using a value higher than three as a threshold produces too few recommendations to be useful.

All elements of the resulting clusters are returned to the user. The technique requires no user interaction besides the definition and the refinement of the input elements, as reflected by "+" in Fig. 8.

Unfortunately, the evaluation of the proposed technique which is included in the paper shows that the benefits of using change clusters are relatively small: the analysis of almost 12 years of software change data for a total of seven different open-source systems showed that fewer than 12% of the studied changes could have benefited from finding elements relevant to the change using change clusters.

Trifu [40] proposes an approach that uses *static dataflow information* to determine the *concern skeleton*—a data-oriented abstraction of a feature. The analyzed program is represented as a *concern graph* whose nodes are variables found in the source code and whose edges are either *dataflow relationships* that capture value transfer between variables or *inheritance relationships* that insure consistent handling of variables defined in polymorphic methods. A path between two variables indicates that the start variable is used to derive the value of the end variable.

The approach treats a feature as an implementation of functionality needed to produce a given set of related values. It receives as input a set of variables that store key results produced by the feature of interest—*information sinks*—and computes a *concern skeleton* which contains all variables in the concern graph that have a path to one of the information sinks. The approach can be optionally provided with an additional set of input variables—*information sources*—that act as cutting points for the incoming paths leading to an information sink. That is, the computed concern skeleton includes only portions of the paths from the given information sources to the given information sinks. The computed concern skeleton is returned to the user.

The approach provides some help in identifying the input set of information sinks by computing a *reduced concern graph* in which certain variables are filtered out (e.g., those that have no incident edges in the concern graph). Still, identifying information sinks is not a trivial task which involves semantic knowledge about what the system does. Also, the user has to do the mapping from variables of the resulting concern skeleton to program statements that use them. Thus, the technique relies on extensive user interaction, as indicated by "+ + +" in Fig. 8.

4.1.2 Guided Output

Robillard [30] leverages static program dependence analysis to find elements that are related to an initial *set of interest* provided by the user. The analyzed program is represented as a PDG whose nodes are functions or data fields and edges are function calls or data access links. Given an input *set of interest*—a set of functions and data fields that the user considers relevant to the feature of interest, the system explores their neighbors in the dependency graph and scores them based on their *specificity*—an element is specific if it relates to few other elements, and *reinforcement*—an element is reinforced if it is related to other elements of interest. For the example in Fig. 1, if the initial set of interest contains elements #3 and #4, reinforcement of element number #7 is high as two of its three connections are to elements of interest. Reinforcement of element #1 is even higher, as its sole connection leads to an element of interest. Yet, specificity of element #7 is lower than that of element #1 since the former is connected to three elements whereas the latter—just to one.

The set of all elements related to those in the initial set of interest is scored and returned to the user as a sorted *suggestion set*. The user browses the result, adds additional elements to the set of interest and reiterates. The amount of the required user interaction in this approach is moderate, as indicated by "++" in Fig. 8: the technique itself only browses the direct neighbors of the elements in the input *set of interest* while the user is expected to extend this set interactively, using the results generated by the previous step.

Saul et al. [35] build on Robillard's technique [30]) and introduce additional heuristics for scoring program methods. The proposed approach consists of two phases: in the first, a set of potentially relevant methods is calculated for an input method of interest. These are the union of caller and callee methods ("parents" and "children"), methods called by the caller functions ("siblings") and methods that call the callee methods ("spouses"). For example, for the element #4 in Fig. 1, elements #2, #3, #7, and #8 are potentially relevant.

The calculated set of potentially relevant methods is then scored using the HITS web mining algorithm (see Sect. 2.4) based on their "strength" as *hubs* (methods that aggregate functionality, i.e., call many other methods) or *authorities* (methods that largely implement functionality without aggregating). The calculated authority score is used to rank the results returned by the algorithm. That is, a method gets a high score if it is called by many high-scored hub methods. In our example, element

#7 has a lower score than #4, because the former is called only by method #8 which is a low-scored hub method as it calls only one method. Element #4 has a higher score because (1) it is called by both elements #7 and #8, and (2) element #7 has a higher hub score as it calls two methods rather than one.

Similar to [30], the technique requires a moderate amount of user interaction, as indicated by "++" in Fig. 8.

Marcus et al. [23,24] introduce one of the first approaches for using IR techniques for feature location. The approach is based on using domain knowledge embedded in the source code through identifier names and internal comments.

The analyzed program is represented as a set of text documents describing software elements such as methods or data type declarations. To create this set of documents (corpus), the system extracts identifiers from the source code and comments, and separates the identifiers using known code styles (e.g., the use of underline "_" to separate words). Each software element is described by a separate document containing the extracted identifiers and translated to LSI space vectors (see Sect. 2.2) using identifiers as terms.

Given a natural language query containing one or more words, identifiers from the source code, a phrase or even short paragraphs formulated by the user to identify a feature of interest,[4] the system converts it into a document in LSI space, and uses the similarity measure between the query and documents of the corpus in order to identify the documents most relevant to the query.

In order to determine how many documents the user should inspect, the approach partitions the search space based on the similarity measure: each partition at step $i + 1$ is made up of documents that are closer than a given threshold α to the most relevant document found by the user in the previous step i. The user inspects the suggested partition and decides which documents are part of the concept. The algorithm terminates once the user finds no additional relevant documents in the currently inspected partition and outputs a set of documents that were found relevant by the user, ranked by the similarity measure to the input query.

For the example in Fig. 1, assume that similarities between documents and a query are calculated as specified in Sect. 2.2 and summarized in Table 1. That is, only terms from method names (and not from method bodies) are used. Under this setting, if α is set to 0.3, the first partition will contain only document d_2 and the second—only d_1. No other document is within 0.3 of d_1 and thus the algorithm will terminate and output d_1 and d_2.

The technique requires no user interaction besides the definition and the refinement of the input query, and thus is marked with "+" in Fig. 8.

Poshyvanyk et al. [26] extend the work of Markus et al. [23, 24] with formal concept analysis (see Sect. 2.1) to select most relevant, descriptive terms from the ranked list of documents describing source code elements. That is, after the

[4] Several approaches, e.g., [4, 9], address the problem of input query definition. They consider not only the query but also related terms when evaluating the document models. As discussed earlier, these approaches are out of the scope of this chapter.

documents are ranked based on their similarity to the input query using LSI, as in [23, 24], the system selects the first n documents and ranks all terms that appear *uniquely* in these documents. The ranking is based on the similarity between each term and the document of the corpus, such that the terms that are similar to those in the selected n documents but not to the rest are ranked higher. Terms that are similar to documents not in the selected n results are penalized because they might be identifiers for data structures or utility classes which would pollute the top ranked list of terms. For the example in Fig. 1, given the LSI ranking with respect to the *automatic save file* query shown in Table 1, if n is set to 2, documents d_1 and d_2 are selected. The unique terms in these are "automatic," "do," and "run," all ranked high as they are not similar to any of the terms in the rest of the documents.

After the unique terms are ranked, the system selects the top k terms (attributes) from the first n documents (objects) and applies FCA (see Sect. 2.1) to build the set of concepts. For the three terms in our example, the concepts are ($\{d_1, d_2\}$, {`automatic, do`}), and ($\{d_1\}$, {`automatic, do, run`}). The terms describe the resulting documents. The user can inspect the generated concepts—the description and links to actual documents in the source code—and select those that are relevant. Similar to [23, 24], the technique requires a low amount of user interaction, as indicated by "+" in Fig. 8.

Shao et al. [36] introduce another approach that extends the work of Marcus et al. [23, 24] by completing the LSI ranking with static call graph analysis. Each method of the analyzed program is represented by a document containing its identifiers. After the LSI rank for each document with respect to the input query is calculated, the system builds a set of methods corresponding to documents ranked above a certain threshold and computes a set of all callers and callees of these methods. The LSI score of the elements in the computed set is augmented to represent their call graph proximity to one of the methods ranked high by LSI. The algorithm outputs a list of all methods organized in a descending order by their combined ranking. For the example in Fig. 1, element #3 is ranked low by LSI with respect to the query "*automatic save file*" (−0.2034 in Table 1). However, it is called by element #1 which has a high LSI rank (0.6319 in Table 1). Thus, the score of element #3 will be augmented and it will be ranked higher.

The technique requires no user interaction except defining and refining the input query describing the feature of interest, as indicated by "+" in Fig. 8.

Hill et al. [18] combine call graph traversal with the *tf-idf*-based ranking (see Sect. 2.3). The analyzed program is represented as a call graph and a set of text documents. Each document corresponds to a method of the program and includes all identifiers used in the method. The user provides an initial query that describes the feature, a seed method from which the exploration starts, and the exploration depth which determines the neighborhood to be explored (i.e., a maximal distance of explored methods from the seed).

Starting from the input seed method, the system traverses the program call graph and calculates the relevance score of each explored method by combining the following three parameters: (1) the *tf-idf* score of the identifiers in the method name; (2) the *tf-idf* score of the identifiers in the method body; and (3) a binary

parameter specifying whether the method is from a library or part of the user code. If the score of a method is higher than a preset relevance threshold, the method is marked as relevant. If the score is higher than a preset exploration threshold (which is usually lower than the relevance threshold) and the distance of the element from the seed is lower than the exploration depth, the system continues exploring the neighborhood of this element. Otherwise, the element becomes a "dead-end," and its neighborhood is not explored. When there are no additional elements to explore for the given exploration depth, the system outputs the call-graph neighborhood of the seed method in which all elements are scored and relevant elements are marked.

For the example in Fig. 1, if the element #1 is used as a seed and the exploration depth is set to 3, all elements other than #2 can be potentially explored. For the sake of the example, we disregard the terms that appear in method bodies and assume that the program does not use any binary methods. In such a case, the calculated score of element #3 is based on the *tf-idf* similarity of the method name to the input query—0.12 for the input query "*automatic save file*," as shown in Sect. 2.3. Thus, setting the exploration threshold above this value results in not exploring the part of the graph starting with element #1, and thus no elements are returned to the user. The exploration threshold of up to 0.12 results in further exploration of the call graph.

The relevance threshold specifies which of the explored elements are considered relevant. Both relevance and exploration thresholds are set empirically, based on the experience with programs under analysis. The technique requires no user interaction besides the definition and the refinement of the input feature description and seed method, and thus is marked with "+" in Fig. 8.

Chen et al. [7] present a technique for retrieving lines of code that are relevant to an input query by performing textual search on the cvs comments associated with these lines of code. The analyzed program is represented as a set of lines for a newest revision of each file. The system examines changes between subsequent versions of each file using the *cvs diff* command and associates the corresponding comment with each changed line. It stores all associated cvs comments for each line of a file in a database and retrieves all lines whose cvs comments contain at least one of the input query's words. The results are scored to indicate the quality of the match: the more query words appear in the comment, the higher is the score. In addition, the system searches the source code to find lines containing at least one of the query's words. It outputs a sorted list of files so that those with the highest number of matches appear first. Within each file, a sorted list of all lines that either match the query or are associated with a cvs comment that matches it is presented.

The technique is largely automated and requires no user interaction other than providing and possibly refining the input query, as indicated by "+" in Fig. 8.

4.2 Dynamic Feature Location Techniques

In this section, we describe techniques that rely on program execution for locating features in source code. The majority of such techniques address the feature

location task for sequentially executed programs, thus we focus the section on those techniques. We note that some of the described approaches have been extended, e.g., [1, 13], to handling distributed and multi-threaded systems as well.

4.2.1 Plain Output

Widle et al. [42] introduced one of the earliest feature location techniques taking a fully dynamic approach. The main idea is to compare execution traces obtained by exercising the feature of interest to those obtained when the feature of interest is inactive. Towards this end, the program is instrumented so that the components executed on a scenario/test case can be identified. The granularity of components, e.g., methods or lines of code, is defined by the user. The user specifies a set of test cases that invoke each feature. The system runs all input test cases and analyzes their execution traces, identifying *common* components—executed by *all* test cases. In addition, for each feature, it identifies (1) *potentially involved* components—executed by *at least one* test case of the feature; (2) *indispensably involved* components—executed by *all* test cases of the feature; and (3) *uniquely involved* components—executed by *at least one* test case of the feature and not executed by *any* test case of the other features. The system outputs sets of potentially involved, indispensably involved and uniquely involved components for each feature, as well as the set of all common components.

For the example in Fig. 1, the execution trace of the *automatic save file* feature can be compared to that of the *manual save file* feature. In this case, elements #3, #5, and #6 are considered *common*, since the *automatic save file* feature relies on the execution of *manual save file* and, thus, these methods are executed in both scenarios. Element #1 is considered *uniquely involved* as it is executed by the *automatic save file* feature only.

Since the user is required to define two sets of scenarios for each feature—those that exercise it and those that do not, the technique requires heavy user involvement and we assess it as "+ + +" in Fig. 8.

Wong et al. [43] present ideas similar to [42]. Its main contribution is in analyzing data flow dependencies in addition to the control flow (method calls) and in presenting a user-friendly graphical interface for visualizing features.

Eisenbarth et al. [14] attempts to address one of the most significant problems of dynamic approaches discussed above—the difficulty of defining execution scenarios that exercise exactly one feature. Their work relies on the assumption that execution scenarios can implement more than one feature and a feature can be implemented by more than one scenario. The work extends [42] with FCA (see Sect. 2.1) to obtain both computation units for a feature as well as the jointly and distinctly required computation units for a set of features.

The analyzed program is represented by an instrumented executable and a static program dependence graph whose nodes are methods, data fields, classes, etc. and whose edges are function calls, data access links, and other types of relationships obtained by static analysis. While in general the technique is applicable

to computation units on any level of granularity, the approach is implemented and evaluated for method-level components. The system first executes all given input scenarios, each of which can invoke multiple features. Optionally, users can identify special *start* and *end* scenarios whose components correspond to startup and shutdown operations and are excluded from all executions.

Users select a subset of execution scenarios they wish to investigate. Then, the approach uses FCA (see Sect. 2.1), where computation units are objects, scenarios are attributes and relationships specify whether a unit is executed when a particular scenario is performed, to create a concept lattice. Based on the lattice, the following information is derived: (1) a set of computation units *specific* to a feature—those used in all scenarios invoking the feature, but not in other scenarios; (2) a set of computation units *relevant* to a feature—used in all scenarios invoking the feature, and possibly in other scenarios; (3) a set of computation units *conditionally specific* to a feature—those used in some scenarios invoking the feature, but not in scenarios that do not invoke the feature; (4) a set of computation units *conditionally relevant* to a feature—those used in some scenarios invoking the feature, and possibly in other scenarios that do not invoke the feature; and (5) a set of computation units *irrelevant* to a feature—those used only in scenarios that do not invoke the feature. In addition, for each feature, the system builds a *starting set* in which the collected computation units are organized from more specific to less. It also builds a subset of the program dependency graph containing all transitive control flow successor and predecessors of computation units in the starting set (i.e., method callers and callees). The graph is annotated with features and scenarios for which the computation units were executed.

The user inspects the created program dependency graph and source code in the order suggested by the starting set, in order to refine the set of identified computation units for a feature by adding and removing computational units. During the inspection, the system also performs two further analyses to assist with the call graph inspection: *strongly connected component analysis* and *dominance analysis*. The former is used for identifying cycles in the dependency graph. If there is one computation unit in the cycle that contains feature-specific code, all computation units of the cycle are related to the feature because of the cycle dependency. The purpose of the latter is to identify computation units that must be executed in order to reach one of the computation units containing feature-specific code. All such computation units are related to the feature as well.

At the end of the process, a set of components deemed relevant for each feature is generated. Even though the technique attempts to assist the user in defining input scenarios, the required level of user interaction in defining the scenarios, selecting the order in which the scenarios are processed, as well as interactively inspecting and refining the produced result is still high, as indicated by “+ + +” in Fig. 8.

Koschke et al. [20] extend the work of Eisenbarth et al. [14] by considering statement-level rather than method-level computation units.

Asadi et al. [2] propose an approach which combines IR, dynamic-analysis, and search-based optimization techniques to locate *cohesive* and *decoupled* fragments of traces that correspond to features. The approach is based on the assumptions that

methods responsible for implementing a feature are likely to share some linguistic information and be called close to each other in an execution trace.

For an input set of scenarios that exercise the features of interest, the system collects execution traces and prunes methods invoked in most scenarios (e.g., those related to logging). In addition, it compresses traces to remove repetition of one or more method invocations and keeps one occurrence of each method. Next, it tokenizes each method's source code and comments, removing special characters, programming language keywords and terms belonging to a stop-word list for the English language (e.g., "the", "is", "at"). The remaining terms are tokenized separating identifiers using known coding styles. The terms belonging to each method are then ranked using the *tf-idf* metric (see Sect. 2.3) with respect to the rest of the corpus. For the example in Fig. 1, when considering only terms of the method names, the term `mind` appears in all documents and thus is ranked 0, while the term `controller` appears only in one document (that corresponds to element #8) and thus gets a higher rank—0.9. The obtained *term-by-document co-occurrence matrix* is transformed to vectors in the LSI space (see Sect. 2.2). A cosine similarity between two vectors in LSI space is used as a similarity measure between the corresponding documents (methods).

Next, the system uses *genetic optimization algorithm* [17]—an iterative procedure that searches for the best solution to a given problem by evaluating various possible alternatives using an *objective function*, in order to separate each execution trace into conceptually cohesive *segments* that correspond to the features being exercised in a trace. In this case, an optimal solution is defined by two objectives: maximizing *segment cohesion*—the average similarity between any pair of methods in a segment, and minimizing *segment coupling*—the average similarity between a segment and all other segments in a trace, calculated as average similarity between methods in the segment and those in different ones. That is, the algorithm favors merging of consecutive segments containing methods with high average similarity.

The approach does not rely on comparing traces that exercise the feature of interest to those that do not and does not assume that each trace corresponds to one feature. Thus, the task of defining the execution scenarios is relatively simple. However, the approach does not provide any assistance in helping the users to understand the meaning of the produced segments and tracing those to the features being exercised in the corresponding scenario; thus, this step requires a fair amount of user interaction. In Fig. 8, we rate this approach as "++".

4.2.2 Guided Output

Eisenberg et al. [15], similar to Eisenbarth et al. [14], present an attempt to deal with the complexity of scenario definition. The approach assumes that the user is unfamiliar with the system and thus should use pre-existing test suites, such as those typically available for systems developed with a test-driven development (TDD) strategy. It accepts as input a test suite that has some correlation between features and test cases (i.e., all features are exercised by at least one test case). Tests that exhibit some part of a feature functionality are mapped to that feature and referred

to as its *exhibiting test set*. Tests which are not part of any exhibiting test set are grouped into sets based on similarity between them and are referred to as the *non-exhibiting test set*.

For each feature, the system collects execution traces obtained by running all tests of the feature's exhibiting test set and generates a *calls set* which lists <*caller*, *callee*> pairs for each method call specified in the collected traces. It then ranks each method heuristically based on the following parameters: (1) *multiplicity*—a relationship between the percentage of tests in the exhibiting test set of the feature that execute the method and the percentage of tests in non-exhibiting test sets that execute that method; (2) *specialization*—the percentage of test sets that exercise the method. (If a method is exercised by many test sets, it is more likely to be a utility method); and (3) *depth*—the call depth (the number of stack frames from the top) of the method in the exhibiting test set compared to that in non-exhibiting test sets. The rationale behind these heuristics is that the exhibiting test set focuses on the feature in the most direct way. This is correlated with the call depth of the methods that implement this feature—the more "directly" a method is exercised, the lower its call depth.

For each feature, both the ranked list of methods and the generated *call set* are returned to the user. The goal of the former is to rank methods by their relevance to a feature, whereas the goal of the latter is to assist the user in understanding *why* a method is relevant to a feature. With respect to the required level of user interaction, we assess the technique as "++" in Fig. 8 because of the effort involved in creating test scenarios, if they are not available.

Poshyvanyk et al. [27] combine the techniques proposed in Marcus et al. [24] and Antoniol et al. [1] to use LSI (see Sect. 2.2) and execution-trace analysis to assist in feature location. The analyzed program is represented by a set of text documents describing software methods and a runnable program instrumented so that methods executed on any scenario can be identified.

Given a query that is formulated by the user to identify a given feature and two sets of scenarios—those that exercise the feature of interest and those that do not, the system first ranks input program methods using LSI. Then, it executes input scenarios, collects execution profiles and ranks each executed method based on the frequency of its appearance in the traces that exercise the feature of interest versus traces that do not. The final rank of each method is calculated as a weighted sum of the above two ranks. The system outputs a ranked list of methods for the input feature.

For the example in Fig. 1, element #1 is executed only in scenarios that exercise *automatic save file*. Thus, its LSI score (0.6319, as calculated in Table 1) will be increased, while the score of element #5 (0.2099, as calculated in Table 1) will be decreased to reflect the fact that it is executed in both scenarios that exercise the *automatic save file* feature and those that do not.

Similar to other dynamic approaches, this approach requires an extensive user involvement for defining scenarios that exercise the feature of interest and those that do not and, therefore, we assess the level of the necessary user interaction for this technique as "+ + +" in Fig. 8.

Liu et al. [22], similar to Poshyvanyk et al. [27], combine the use of LSI and execution-trace analysis. However, this work proposes operating on a *single* trace rather than on multiple traces that exercise/do not exercise the feature of interest.

Given a query that is formulated by the user to identify a feature of interest and a *single* scenario capturing that feature, the system executes the input scenario and ranks methods executed in the scenario using LSI with respect to the input query as in [24]. A ranked list of executed methods is returned to the user. For our example in Fig. 1, a scenario that executes the *automatic save file* feature invokes elements #1, #3, #6, and #7. These elements are returned to the user together with their LSI ranking, shown in Table 1.

Since the user is only required to provide a single scenario that exercises each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as "+" in Fig. 8.

Rohatgi et al. [33] present a technique that is based on dynamic program analysis and static program dependence graph analysis. The technique operates on a class level, where the analyzed program is represented by an instrumented executable and a static program class dependency graph whose nodes are classes and whose edges are dependency relationships among these classes such as method calls, generalization, and realization.

As input, the system obtains a set of scenarios that invoke the features of interest. It executes all input scenarios, collects execution profiles on a class level, and uses *impact analysis* to score the relevance of the classes to the feature of interest: classes that impact many others in the system are ranked low as these classes are likely not feature-specific but rather "utility" classes implementing some core system functionality. The technique outputs a set of classes produced by the dynamic trace analysis, ranked by their relevance as calculated using impact analysis.

We assess the level of the necessary user interaction for this technique as "++" in Fig. 8 because it requires only a set of scenarios that invoke the features of interest and not those that don't.

Eaddy et al. [12] present the PDA technique called *prune dependency analysis* which is based on the assumption that an element is relevant to a feature if it should be *removed* or otherwise *altered* if the feature is removed from the program. The program is represented as a *program dependence graph* whose nodes are classes and methods, and whose edges are method invocations, containment relationships between a class and its methods, or inheritance relationships between classes. The system calculates the set of all elements affected by removing at least one element from the seed input set. For the example in Fig. 1, removing element #2 requires removing or altering element #4 that initiates a call to it in order to avoid compilation errors. Thus, element #4 is related to the feature that involves execution of element #2. Removing element #4 requires removing elements #7 and #8. The latter does not trigger any additional removals.

Furthermore, the work suggests combining the proposed technique with existing dynamic- and IR-based feature location approaches to achieve better accuracy. The dynamic feature location can use the approaches proposed in [15, 42] or others. These either produce a ranked set of methods, as in Eisenberg et al. [15] or an

unsorted list of relevant elements, as in Wilde et al. [42]. In the latter case, an element is assigned the score 1 if it is executed *only* by scenarios exercising the feature of interest, or 0 otherwise. The IR-based feature location uses the approach of Zhao et al. [44]: program elements are ranked with respect to feature descriptions (extracted from requirements) using the *vector space model*. It calculates the cosine of the distance between the corresponding vectors of terms, each of which first weighted using the *tf-idf* metric.

For each software element, the resulting score is calculated by normalizing, weighing and adding the similarity scores produced by the IR and the dynamic techniques, as in Poshyvanyk et al. [27]. Then, similar to Zhao et al. [44], the system applies a threshold to identify highly relevant elements. These are used as input to the *prune dependency analysis* which produces the set of additional relevant elements. The resulting set, ranked by the combination of scores produced by IR and dynamic techniques, is returned to the user.

For our example in Fig. 1, elements #1 and #2 are ranked high by the vector space model for the query "*automatic save file*." Since element #1 is executed only by scenarios that exercise the *automatic save file* feature, it is also ranked high by a dynamic analysis-based technique. Prune dependency analysis uses these two as the input seed set and adds elements #4, #7, and #8, so the result becomes {#1, #2, #4, #7, #8}. Since the technique requires two sets of scenarios for each feature—those that exercise it and those that do not, we assess the level of the necessary user interaction for this technique as "$+ + +$" (see Fig. 8).

Revelle et al. [29] propose improving the feature location accuracy by combining Similar to Liu et al. [22], the proposed system obtains as input a single scenario that exercises the feature of interest and a query that describes that feature. It runs the scenario and constructs a call graph from the execution trace, which is a subgraph of the static call graph and contains only the methods that were executed. Next, the system assigns each method of the graph a score using one of the existing web-mining algorithms—either HITS (see Sect. 2.4) or the PageRanked algorithm developed by Brin and Page [5], which is also based on similar ideas of citation analysis. The system then either filters out low-ranked methods (e.g., if the HITS authority score was used, as in Saul et al. [35]) or high-ranked methods (e.g., if the HITS hub score was used, as high-ranked methods represent common functions). The remaining set of elements is scored using LSI (see Sect. 2.2) based on their relevance to the input query describing the feature. The ranked list of these elements is returned to the user.

For the example in Fig. 1, elements #1, #3, #5, and #6 are invoked when the scenario exercising the *automatic save file* feature is executed. Assuming these elements are scored using HITS authority values, filtering out low-scored methods removes element #1 from the list of potentially relevant elements as its authority score is 0, as shown in Sect. 2.4. The remaining elements, #3, #5 and #6, are scored using LSI with respect to the query "*automatic save file*" (these scores are given in Table 1) and are returned to the user.

Similar to [22], since the user is only required to provide a single scenario for each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as "+" in Fig. 8.

5 Which Technique to Prefer?

As the survey shows, there is large variety in existing approaches and implementation strategies for feature location. We believe that trying to identify a single technique that is superior to the rest would be impractical. Clearly, there is no "silver bullet," and the performance of each technique largely depends on its applicability to the analyzed input programs and the quality of the feature description (*feature intension*) provided by the user. In this section, we discuss considerations and provide explicit guidelines for practitioners who need to choose a particular feature location technique to apply.

The chosen technique should first and foremost be suitable to the program being analyzed: specifically, if the studied program contains no documentation and no meaningful identifier names, IR-based feature location techniques will be unable to achieve high-quality results. Similarly, if the implementation of a feature is spread across several program modules or is hooked into numerous extension points provided by the platform on which the program is built (e.g., invoking methods via event queues), techniques based on program dependency analysis will either be unable to find all elements that relate to the implementation of the feature or will find too many unrelated elements. When program execution scenarios are unavailable or it is cumbersome to produce scenarios that execute a specific set of features (e.g., because the feature of interest is not a functional feature that is "visible" at the user level), dynamic feature location techniques will not be applicable. Figure 9 assesses the surveyed feature location techniques based on the above selection criteria.

For our example in Figs. 1 and 2, program elements have meaningful names ("file" vs. "f" or "property" vs. "prp"). Thus, it is reasonable to choose one of the techniques that rely on that quality, as marked in the corresponding column of Fig. 9. Since the implementation of the Freemind software is asynchronous and relies on event queues to perform method invocation, techniques that analyze call graph dependency might be less efficient. In addition, defining a scenario that triggers the *automatic save file* feature might not be trivial—there is no user operation that directly invokes the automatic save (as opposed to the manual save) functionality. Therefore, techniques that do not require program execution are a better choice which leads us to the approaches in Shepherd et al. [38], Marcus et al. [23, 24], or Poshyvanyk et al. [26].

With respect to the quality of a feature intent provided by the user, IR-based techniques are usually most sensitive to the quality of their input—the query that describes the feature of interest. The results produced by these techniques are often as good as the query that they use. Input query definition and the user assistance during that process are further discussed by [4, 9] and others. Techniques based on

		Technique	Strongly Coupled Implementation	Meaninful Names	Change Histories	Execution Scenarios
Static	Plain	**Chen et al. [8]**	√			
		Walkinshaw at al. [41]	√			
		Shepherd et al. [38] (Find-concept)		√		
		Zhao et al. [44] (SNIAFL)	(√)	√		
		Robillard et al. [31]			√	
		Trifu [40]	(√)			
	Guided	**Robillard [30]** (Suade)	√			
		Saul et al. [35]	√			
		Marcus et al. [23, 24]		√		
		Poshyvanyk et al. [26]		√		
		Shao et al. [36]	(√)	√		
		Hill et al. [18] (Dora)	√	√		
		Chen et al. [7]			√	
Dynamic	Plain	**Wilde et al. [42]** (Sw. Reconnaissance)				√
		Wong et al. [43]				√
		Eisenbarth et al. [14]	(√)			√
		Koschke et al. [20]	(√)			√
		Asadi et al. [2]		√		√
	Guided	**Eisenberg et al. [15]**				√
		Poshyvanyk et al. [27]		√		√
		Liu et al. [22] (SITIR)		√		√
		Rohatgi et al. [33]				√
		Eaddy et al. [12] (Cerberus)	√	√		√
		Revelle et al. [29]		√		√

Fig. 9 Criteria for selecting a feature location technique

comparing dynamic execution traces are also sensitive to the nature of their input—if execution scenarios do not cover all aspects of the located feature, the accuracy of the feature location will likely be low.

The approaches also differ in the required level of user interaction (see the last column of Fig. 8). We assess the level of user interaction based on the *effort* that the user has to invest in *operating* the technique. This includes the effort involved in defining the input feature intension (e.g., a set of scenarios exercising the features of interest), interactively following the location process (e.g., filtering intermediate results produced by the technique) and interpreting the produced results (e.g., mapping retrieved variables to the code statements that use them).

Since more highly automated techniques are easier to execute, their “barrier to entry”—the effort required to produce the initial approximation of the result—is lower and thus their adoption is easier. On the other hand, the techniques that require more user interaction are usually able to produce better results because they harvest this “human intelligence” for the feature location process.

Furthermore, automated techniques could be a better choice for the users that seek an “initial approximation” of the results and are able to complete them manually since they are familiar with the analyzed code. On the other hand, users that cannot rely on their understanding of the analyzed code should probably choose a technique that is more effective at producing relevant results, even though operating such a technique requires a more intensive investment of time and effort.

6 Summary and Conclusions

In this chapter, we provided a detailed description of 24 feature location techniques and discussed their properties. While all of the surveyed approaches share the same goal—establishing traceability between a specific feature of interest that is specified by the user and the artifacts that implement that feature, their underlying design principles, their input, and the quality of the results which they produce differ substantially. We discussed those in detail and identified criteria that can be used when choosing a particular feature location technique in a practical setting. We also illustrated the techniques on a common example in order to improve the understandability of their underlying principles and implementation decisions.

Even though the area of feature location is mature, there is variety in existing techniques, which is caused by the common desire to achieve high accuracy: automatically find a high number of relevant elements (high *recall*) while maintaining a low number of false-positive results (high *precision*). As discussed in Sect. 5, since there is no optimal technique, each of the approaches proposes heuristics that are applicable in a particular context, making the technique efficient in these settings.

Feature Location for SPLE. In the context of product line engineering, identifying traceability between product line features and product artifacts that realize those features is an essential step towards capturing, maintaining, and evolving well-formed product line systems. Traceability reconstruction is also an important step when identifying product line architectures in existing implementations.

Each of the existing feature location techniques can be used for detecting features of products that belong to a product family. Feature location is done while treating these products as singular independent entities. Yet, considering families of *related* products can provide additional input to the feature location process and thus improve the accuracy of the techniques by considering product line commonalities and variations.

When considering a specific feature that exists only in some products of the family, comparing the code of a product that contains the feature to the code of the one that does not can partition the code into two parts: *unique* to the product and *shared*. This partitioning can help detect relevant elements with higher accuracy because it limits the results to the elements of the *unique* part where the feature of interest is located. For example, it can be used to filter out irrelevant elements (those that belong to the shared parts of the code) from the program execution trace analyzed by Liu et al. [22].

The above partitioning can also improve scoring and traversal mechanisms employed by existing feature location techniques when *searching* for these relevant elements. For example, it can be used for augmenting the score calculation formula used by Hill et al. [18] so that the score of elements belonging to the shared parts of the code is decreased while the score of those in the unique parts is increased, as shown in [34]. This affects the call graph traversal process and the ability of the algorithm to reach the desired elements, while avoiding passes that lead to

false-positive results. More complex partitioning, obtained by comparing multiple products to each other, can provide even better solutions.

In addition, it might be interesting to develop methods for incremental analysis of product lines, where the traceability links obtained for one variant may be carried over to the next variant. This will prevent unnecessary re-analysis and leverage the effort and human intelligence invested in one product for more efficient feature location in others. We explore this and other directions in our ongoing work.

References

1. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: an epidemiological metaphor. IEEE TSE **32**, 627–641 (2006)
2. Asadi, F., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G.: A heuristic-based approach to identify concepts in execution traces. In: Proc. of CSMR'10, pp. 31–40, 2010
3. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman, Boston (1999)
4. Bai, J., Song, D., Bruza, P., Nie, J.Y., Cao, G.: Query expansion using term relationships in language models for information retrieval. In: Proc. of CIKM'05, pp. 688–695, 2005
5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proc. of WWW7, pp. 107–117, 1998
6. Brooks, F.P. Jr.: No silver bullet essence and accidents of software engineering. IEEE Comput. **20**, 10–19 (1987)
7. Chen, A., Chou, E., Wong, J., Yao, A.Y., Zhang, Q., Zhang, S., Michail, A.: CVSSearch: Searching through source code using CVS comments. In: Proc. of ICSM'01, 2001
8. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: Proc. of IWPC'00, pp. 241–249, 2000
9. Cleary, B., Exton, C., Buckley, J., English, M.: An empirical analysis of information retrieval based concept location techniques in software comprehension. J. Empir. Software Eng. **14**, 93–130 (2009)
10. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley Longman, Boston (2001)
11. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process **25**(1), 53–95 (2013)
12. Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.G.: CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Proc. of ICPC'08, pp. 53–62, 2008
13. Edwards, D., Simmons, S., Wilde, N.: An approach to feature location in distributed systems. J. Syst. Software **79**, 57–68 (2006)
14. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE TSE **29**, 210–224 (2003)
15. Eisenberg, A.D., De Volder, K.: Dynamic feature traces: finding features in unfamiliar code. In: Proc.of ICSM'05, pp. 337–346, 2005
16. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, New York (1999)
17. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman, Boston (1989)
18. Hill, E., Pollock, L., Vijay-Shanker, K.: Exploring the neighborhood with dora to expedite software maintenance. In: Proc. of ASE'07, pp. 14–23, 2007

19. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM **46**, 604–632 (1999)
20. Koschke, R., Quante, J.: On dynamic feature location. In: Proc. of ASE'05, 2005
21. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process. **25**(2–3), 259–284 (1998)
22. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proc. of ASE'07, 2007
23. Marcus, A.: Semantic-driven program analysis. In: Proc. of ICSM'04, pp. 469–473, 2004
24. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: Proc. of WCRE'04, pp. 214–223, 2004
25. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering : Foundations, Principles, and Techniques. Springer, New York (2005)
26. Poshyvanyk, D., Marcus, A.: Combining formal concept analysis with information retrieval for concept location in source code. In: Proc. of ICPC'07, pp. 37–48, 2007
27. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE TSE **33**, 420–432 (2007)
28. Qin, T., Zhang, L., Zhou, Z., Hao, D., Sun, J.: Discovering use cases from source code using the branch-reserving call graph. In: Proc. of APSEC'03, pp. 60–67, 2003
29. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: Proc. of ICPC'10, pp. 14–23, 2010
30. Robillard, M.P.: Automatic generation of suggestions for program investigation. In: Proc. of ESEC/FSE-13, pp. 11–20, 2005
31. Robillard, M.P., Dagenais, B.: Retrieving task-related clusters from change history. In: Proc. of WCRE'08, pp. 17–26, 2008
32. Robillard, M.P., Shepherd, D., Hill, E., Vijay-Shanker, K., Pollock, L.: An empirical study of the concept assignment problem. Tech. Rep. SOCS -TR-2007.3, School of Computer Science, McGill University (2007)
33. Rohatgi, A., Hamou-Lhadj, A., Rilling, J.: An approach for mapping features to code based on static and dynamic analysis. In: Proc. of ICPC'08, pp. 236–241, 2008
34. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. In: Proc. of ASE'12, pp. 242–245, 2012
35. Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending random walks. In: Proc. of FSE'07, pp. 15–24, 2007
36. Shao, P., Smith, R.K.: Feature location by IR modules and call graph. In: Proc. of ACM-SE 47, pp. 70:1–70:4, 2009
37. Shepherd, D., Pollock, L., Vijay-Shanker, K.: Towards supporting on-demand virtual remodularization using program graphs. In: Proc. of AOSD'06, pp. 3–14, 2006
38. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proc. of AOSD'07, pp. 212–224, 2007
39. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. **3**(3), 121–189 (1995)
40. Trifu, M.: Improving the dataflow-based concern identification approach. In: Proc. of CSMR'09, pp. 109–118, 2009
41. Walkinshaw, N., Roper, M., Wood, M.: Feature location and extraction using landmarks and barriers. In: Proc. of ICSM'07, pp. 54–63, 2007
42. Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. J. Software Mainten. **7**, 49–62 (1995)
43. Wong, W.E., Horgan, J.R., Gokhale, S.S., Trivedi, K.S.: Locating program features using execution slices. In: Proc. of ASSET'99, pp. 194–203, 1999
44. Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F.: SNIAFL: towards a static noninteractive approach to feature location. ACM TOSEM **15**, 195–226 (2006)