Report on the Feature Location Technique to be Used with SubSystemA

Muslim Chochlov

August 24, 2017

Abstract

As part of the "SystemA, Product-Family, Analysis-Engine Unification Project", the features in SubSystemA module (and their equivalents/clones in other products) have to be extracted to be later unified as a singular code element for re-use. The first step to the unification process is to locate features in SubSystemA. The 4 major types of analysis that are used in feature location are summarized in this document: dynamic, static, textual, and historical. Also, in this report, the characteristics of SubSystemA are presented and the manual feature location of 4 SubSystemA domain engineering features is discussed to gauge how SubSystemA would respond to different types of analysis. The textual analysis was able to locate sub-programs for all of these 4 features with 50.23% average precision and 66.6% recall on average. In one illustrative case, the low recall for a (30.8%) feature was improved by static structural dependencies analysis to locate more relevant sub-programs, reaching a recall of 76.92%. In another illustrative case, using a different type of static structural analysis, 90% of conditional/loop blocks of code were identified for another feature. Following these findings and observations, a semi-automated, two-phase, two-iteration hybrid (textual and static) feature location technique was proposed to assist with the process of feature location in SubSystemA.

Abbreviations

API ..... Application programming interface

FEA ..... Finite Element Analysis

FL ..... Feature location

FLT ..... Feature location technique

I/O ..... Input/output

IR ..... Information retrieval

ITS ..... Issue tracking system

LDA ..... Latent Dirichlet allocation

LSI ..... Latent semantic indexing

NLP ..... Natural language processing

PM ..... Pattern matching

SDG ..... Structural dependency graph

SOA ..... Service oriented architecture

$\mathbf{TFLT_{IR}}$ ..... Textual IR-based FLT

VCS ..... Version control system

VSM ..... Vector space model

1 Introduction

SystemA is a versatile software package that can be used to model many riser types including flexible risers, top tension risers, pipelines, and others. SystemA is built from two software systems: the GUI part and the analysis part. The analysis part is the business layer software sub-system, that allows for structural analysis of sub-sea structures (e.g. risers) using finite element analysis (FEA).

SystemA is also the parent of a family of similar software products (variants): SystemA first diverged into SystemB (less flexible, longer risers) and SystemC software products. The "product family" was created using code cloning from one variant to the other (from parent product to child product and vice versa). Because of the introduced redundancy, the effort involved in maintenance and development of this software product family has increased [4]. To address this problem, our industrial partner have decided that the code, implementing the functionality in the business layer of all variants (e.g. system analysis part in SystemA), should be re-unified in a single code base for systematic maintenance and re-use.

The goal of the current project is to identify and re-factor a subset of the functionality located in SubSystemA, the time domain analysis module (a module is a sub-system in this context) of the SystemA analysis part, and the equivalent modules in SystemB and SystemC. According to the current strategy, this should be accomplished in two steps:

1. Locate the features in SubSystemA.
2. Find the clones of these features in SystemB and SystemC products.

The goal of this report is to propose the semi-/automated technique for step 1: locate the features in SubSystemA.

In the context of this project, a feature refers to a functionality that is implemented in a software system's source code and is characterized as:

- Abstract (a feature is not a single source code component, but a higher level concept that typically cuts across several software components)
- Functional in that end-users see them as real world sub-goals of the system, that are implemented in the software systems
- Observable (removing or adding a feature - and hence its implementing code - will result in changes to the software's product state or behaviour).

As the characterization above implies, features are implemented in source code, using one or more source code components, where a source code component is a specific piece of code such as a statement, variable, sub-program, class, file, etc. The entire set of source code components that implement the feature is referred to as the extent of a feature. Feature location (FL) then is mapping a feature to source code components.

Manual feature location can be difficult [12, 13, 16, 23, 24] and costly [3, 5]. To address the difficulty of manual feature location and the costs that arise

because of that difficulty, research has been directed at applying some degree of automation to this process. A feature location technique (FLT) is a semi-/automated approach to feature location. According to Dit et al. (the largest review of FLTs to date), there are 4 major types of FLTs: dynamic, static, textual, and historical [10]. (A fifth type: the combinations of these FLTs, are called hybrid FLTs in this report.) This classification is used in this report to describe FLTs.

It was shown recently [1, 7, 17] that the configuration of a FLT and the characteristics of a software system to which such an FLT is applied could impact the effectiveness of a FLT. To gauge how SubSystemA would respond to different types of FLTs, SubSystemA was analysed and manual FL was performed to locate the 4 domain engineering features in the code. In summary, the results showed that the dynamic source code analysis was able to locate source code components for 2 of these features, whereas the textual analysis worked in all 4 cases. The two illustrative examples showed that the textual analysis could be improved by static source code analysis.

As a result, a semi-automated two-phase two-iteration hybrid (textual and static) FLT is being proposed. The FLT utilizes textual source code analysis to locate the relevant sub-programs and then uses the static analysis of structural dependency in source code to locate adjacent sub-programs and conditional/loop code blocks.

This report is organized as follows: Section 2 provides a quick overview of the state-of-the-art in FLTs. Section 3 presents the SubSystemA and its characteristics. Section 4 describes the process and analysis of the manual FL of the 4 features in SubSystemA. In Section 5 a FLT is proposed, based on the findings from our manual FL exercise, to be used with SubSystemA. Section 6 concludes this report.

2 Overview of Feature Location Techniques

As mentioned in Section 1, according to Dit et al. [10] there are 4 major types of FLTs: dynamic, static, textual, and historical. The dynamic FLTs leverage runtime data of a software system, static FLTs leverage the structural data of source code, textual FLTs leverage the meaningful lexical data that is available in source code, and historical FLTs leverage the auxiliary data associated with the software system's source code.

2.1 Dynamic Techniques

In dynamic FLTs, scenarios are used to run a software system in such a way that certain features of the software system are/are not triggered as part of the software system's execution. These scenarios are usually test cases or use cases, but can incorporate other input types: for example, external files or parameters of a software system. Before the execution, a software system is instrumented in such a way that runtime information can be collected during the execution. This

runtime information is called execution trace information. The execution traces store an ordered set of source code components (often these are sub-program calls) as they appear during the software system's execution. Therefore, for each scenario there are corresponding execution traces and, hence, the features of the scenario can be mapped to the source code components from the execution traces. However, only a small fraction of the source code components in an execution trace are relevant to a feature [25]. (For example, a menu tree, used to get to the feature of interest would be non-specific to the feature and probably highly similar to that of several other features.) Dynamic FLTs employ different strategies to identify that relevant fraction.

One of the first dynamic FLTs was introduced by Wilde et al. [25]. The authors implemented scenarios as test cases targeted at specific features to capture relevant sets of source code components. According to the authors, there exists a set of shared (e.g. utilities) source code components (executed in several scenarios), a set of mandatory components (executed in all scenarios that exercise a feature), a set of feature-relevant components (executed in at least 1 scenario that exercises a feature), and a set of feature-unique components (which are only exercised by the scenarios exercising that feature and no other scenario). To identify the feature-unique source code components, the authors created 252 test cases: half of which triggered a feature and the other half does not. The source code components are feature-unique if they appear in the execution traces of the test cases that turn the feature on, but do not appear in the execution traces of any of the test cases that don't turn the feature on. The set of 252 test cases seems to be very large, and after further experiments it was concluded that only 12 test cases were needed on average to produce the same result (the set of feature-unique source code components for a given feature). The authors stated that these 12 test cases could be reduced even further, down to just two pairs of test cases, if system experts were to be involved when creating these test cases. But this is a limitation of their approach: the generation of judicious test-cases involves careful (i.e. effort intensive), expert consideration.

In a more recent study, Yousefi et al. [26] used a dynamic FLT to locate features in *service oriented architecture* (SOA). In SOA, services (software systems) could be implemented in a variety of languages, could run on a variety of platforms, and communicate with each other and the end-user via messaging. Such an architecture presents additional difficulty to dynamic FLTs, because the execution could be asynchronous, distributed and concurrent. This means that, scenario-triggered, feature-related code can run in several software systems at the same time and interleave with other scenarios currently running. The main contribution of Yousefi et al. [26] was to aggregate execution traces of various software systems related to a specific feature and so can be seen as an extension of Wilde et al.'s [25] work for more modern software systems.

In another study, Heydarnoori et al. [11] applied a dynamic FLT to locate features using the *application programming interface* (API) of a framework. In computer science, a framework is a domain specific environment that facilitates the creation of domain specific software systems. The frameworks usually provide templates, tools, and source code components that are accessible through

the framework's API. Usually frameworks control the execution of a software system, which marks them out as different from more traditional libraries (collections of source code components). The authors used programs, that leverage the framework's API, as execution scenarios. Unlike Wilde et al. [25], to identify feature-unique source code components in execution traces, they marked the beginning and the end of feature-related API calls in these test programs.

In general, the application of dynamic analysis to FLTs showed that this type of analysis could be effective when locating features in source code. Arguably, the major advantage of such analysis is that given a scenario, that can result in a turned on/off feature, one is guaranteed to find at least a fraction of feature-related source code components. However, there are also fundamental problems with dynamic FLTs:

- The core problem with dynamic FLTs is their inability to locate features that are always executed. In other words, there doesn't exist a scenario that would turn such a feature off. Thus source code components that belong to this feature will always be part of the execution traces.
- The dynamic FLTs seem to be good at locating entry points to a feature, but not the whole extent of a feature [25].
- Though some work has been done on reusing existing test cases for dynamic FLTs [27], the scenarios still need to be created and maintained as the software system evolves. (The ongoing maintenance is probably less of an issue for feature location in legacy code, where feature location is more likely to be a "once-off" activity.) That is, FLTs are supposed to reduce the effort of feature location in source code. The creation and maintenance of scenarios, which is a necessary element of dynamic FLTs, requires additional expert effort, raising the question: how useful are dynamic FLTs in reducing the FL effort?
- The software systems need to be executed to collect execution traces. This could be a concern for software systems with long execution times. The software system has to be executed at least twice to collect the execution traces with and without specific feature-related source code components. Wilde et al. [25] suggest that at least two pairs of test cases (4 test cases) should be used (and 12 test cases on the average). Hence, the execution time will be multiplied by 2 and 6 respectively.

2.2 Static Techniques

Static FLTs analyse structural dependencies of source code components. Since, source code components don't often operate in isolation, but instead communicate with each other and pass information, their interactions could be a support for FL. For example, a developer could start at an interesting source code component and inspect all outgoing sub-program calls, related to that component.

Chen et al. [6] were among the first to apply static structural analysis to FLTs using an "abstract static dependency graph". The graph might represent

either the control or data-flow of a software system. In the first case, the nodes of a graph are sub-programs and the edges are the sub-program calls. In the second case, the nodes are variables and the edges correspond to data transfer. When using the graph to locate the features, a developer has to decide on a starting point (i.e. a graph entry point). In this case he might either start from the "main" sub-program, from any arbitrary node in the graph, or, for example, use information from a change request to reason about the starting point. The static techniques would then further assist a programmer to navigate to relevant code as he walks the graph. For example, the relationships between code components become explicit, and additional visualization techniques might assist by suggesting unexplored paths and showing already selected elements. A developer might choose between a breadth-first and depth-first search strategy for the graph, and decide if bottom-up or top-down feature location is preferable [6]. For example, a change request could be used to indicate that a sub-program A could be a good starting point. A calls two other sub-programs B and C. A developer inspects the latter two sub-programs and concludes that B is relevant, whereas C is not. He might then continue to inspect the neighbourhood of B in a similar manner.

While the approach by Chen et al. [6] facilitates the graph traversal by offering the nodes for inspection and remembering the visited nodes, it still requires much developer's interaction. For example, a developer has to manually inspect all the nodes that are connected to the starting node and to decide on their relevance. A more advanced approach by Robillard et al. [20] suggests the more relevant nodes for inspection by ranking them higher. This ranking is based on two metrics: "specificity" and "reinforcement". For example, the sub-program is more specific if it called by a fewer number of other sub-programs. The sub-program is reinforced if it is called by other sub-programs that were already identified as part of a feature.

Since, static FLTs usually require regular user interaction and a sensible starting point, they are mostly utilized as helper tools or in a combination with other types of FLTs [18, 22]. Static FLTs are useful when finding the extent of a feature or the set of source code components liable to change as part of an impact analysis. The main disadvantage, is the need for the starting point to be selected accurately. Starting from the "main" sub-program could be justified in the case of a program. However, large software system would accordingly have a large graph of components and traversing all the nodes to locate initial relevant ones would likely require significant effort.

2.3 Textual Techniques

Textual FLTs leverage meaningful textual information available directly in source code or in auxiliary sources associated with a software system. In source code this might, for example, be meaningful variable names, meaningful sub-program names, meaningful folder names, the words in comments or the words in output statements. In auxiliary sources it might be, for example, the words in documentation or version control systems.

The user input to textual FLTs is a search query. The search query may consist of search keywords or it may also be a natural language sentence. The search keywords or the words in the natural language sentence are derived from the textual feature descriptions. The textual feature descriptions might come from:

- Documentation of a software system
- Change requests
- Domain knowledge of a system expert (the expert knows how to describe a feature)
- Other textual sources (e.g. team communication textual documents such as emails)

Given a search query, the task of a textual FLT is to return the list of source code components that match the query, based on the similarity between the user query and the meaningful textual information described above. Broadly speaking, to accomplish that, the TFLTs usually have to:

- Represent the source code components using the textual information in source code or in auxiliary sources.
- Match the search query against these textual representations of source code components.

The methods, used in TFLTs to match the search query, might be as simple as pattern matching (PM), or might utilize more sophisticated information retrieval (IR) and natural language processing (NLP) approaches [10]. The IR based approaches seem to be the most popular, because they allow for a reasonable trade-off between the naive simplicity of PM approaches and more complex NLP approaches.

There are several common steps that are usually a part of TFLT_{IR}:

- *Source code partitioning.* In this step, source code is partitioned into source code components of selected granularity. Each source code component is then represented by its textual data in the form of a document.
- *Preprocessing.* During the preprocessing step the data of the textual documents, representing the source code components, is filtered and normalized. Usually there are several preprocessing steps:
  - Filtering: the common words, reserved programming language words (if applied to source code), special symbols, and/or numbers are removed. The common words are usually referred to as "stop words". For example, "a", "the", "is" are stop words. The list of stop words could be different in various implementations. For example, the list of reserved programming language words differs per language: in Java, "if", "for", "public" are common reserved words.

  - Splitting the words: the identifiers could have lengthy names that consist of several words. For example, it is common to encounter the names such as "sort_list_ascending_order" (4 words) or "isVariableInitialized" (3 words). These concatenations have to be split.
  - Normalization and stemming: the words are converted in to lower case for convenience and their stems are extracted. For example, the words "sort", "sorting", and "sorted" have one common stem: "sort".

- *Applying the IR models* The IR models are applied to the textual documents and the documents are organized into a search corpus. Common IR models are vector space model (VSM), latent semantic indexing (LSI), and latent Dirichlet allocation (LDA) [2, 8, 21].

- *Querying.* The search query is transformed into a text document and is matched against the search corpus.

- *Retrieval.* The results (textual documents) are presented in ranked decreasing order, according to their relevance to the search query, and this provides a link links to the associated source code components.

A good example of a textual FLT that utilizes IR (TFLT_{IR}) is a work by Marcus et al. [15]. The authors proposed a TFLT_{IR} that splits the source code into source code components of sub-program level granularity (method level, given the Java context of software systems in their work). The textual data (comments and identifiers), associated with these components was preprocessed: stop words and special symbols were removed and the words were stemmed. The LSI IR model was used to represent and query the textual corpus of these documents and produced a ranked list of the documents that pointed developers to relevant code. .

2.4 Historical Techniques

There are several popular auxiliary data sources that contain historical/evolutionary information about a software system. The most commonly used are:

- Version control systems (VCS): these store and track versions of all the files (not just source code) of a software system.

- Issue tracking systems (ITS): in the context of software engineering, these systems manage and track the execution of change requests.

- Task management systems: these allow developers to organize source code pieces around a task.

- Code review systems: these allow developers to give feedback on code changes (usually via some links with VCSs).

- Informal communication tools: these include email, chat, and other informal tools.

FLTs usually utilize VCSs and ITSs, because of:

- Their meaningful relevant-to-source-code textual data [14].
- Their evolutionary coupling data (leveraging the co-change information of source code components: this mostly applies to VCSs).
- Their ties to source code (in the case of ITSs these are transitive ties, through VCSs).
- Their ubiquity.

Probably the two major applications of VCSs' and ITSs' data are the study of co-changes and the usage of textual data.

The latter usually involves the usage of similar approaches to those used for textual FLTs (see Section 2.3). For example, for the Kayley FLT [19], the authors build a search corpus of textual documents using a combination of the textual data of a change-set (a recorded change in VCSs, also known as a commit), the textual data of source code components, referenced by that change-set (and their structural neighbours), and the textual data of corresponding change request, if available. For example, consider a change-set CS touching sub-programs S_1, S_2, S_3. Sub-program S_1 calls two other sub-programs S_4, S_5. CS has a reference to the change request CR. Then the textual document will be a combination of the CS description, the textual data of S_1, S_2, S_3, S_4, S_5, and the description in CR: in essence, such an approach expands the vocabulary of change-sets. The search query is then matched against the search corpus of such textual documents.

Some advantages of historical FLTs are:

- Distinct, meaningful, auxiliary textual data, relevant to source code components. this can be used as an alternative or in combination with source code data for textual source code analysis.
- Evolutionary information: files that frequently change together are more likely to belong together: possibly belonging to the same feature. This knowledge could be used towards finding the extent of a feature.

The disadvantages are:

- Similarly to textual FLTs in general, the historical FLTs are sensitive to the amount and quality of the textual data in auxiliary sources.
- Additional analysis techniques are required when processing this auxiliary data: when extracting and tying the textual data to the source code and when analysing the evolutionary information.

2.5 The Traits in FL

Currently, there seem to be 3 major trends in research of FLTs [10]:

- The increasing prevalence of hybrid FLTs. Hybrid FLTs were analysed in 52% of the papers identified by Dit et al. [10].
- The prevalence of textual source code analysis, where this type of analysis is used as either standalone textual FLT or as part of a hybrid FLT: 53% of papers from Dit et al. [10] employ textual source code analysis.
- The move towards the evaluation of various configurations of FLTs. Typically FLTs present a "one-size-fits-all" approach and do not study the best-performing configurations. This trend has started to change recently with Biggers et al. [1] studying the influence of LDA parameters and selection of data sources for textual FLTs. In another example, Dit et al. [9] investigated the impact of applying various identifier splitting techniques to textual FLTs. Finally, a more recent study suggests that characteristics of software systems should guide the selection of a FLT [7].

3 SubSystemA: The Time Domain Analysis Module

SubSystemA is a time domain analysis module in SystemA's analysis part. SubSystemA uses FEA over a time series to predict how a sub-sea structure (various risers) will react to the real world environment. FEA is a domain-independent numeric approximation method often used in structural analysis. The idea behind the FEA is to break the complex problem (structure in our case) into smaller sub-problems (finite elements) and to solve these smaller problems in a system of equations that model the entire problem.

In essence, SubSystemA is using the following finite element equation of motions formula:

$$Ku + Du' + Mu'' = F \tag{1}$$

where K is a stiffness matrix, D is a damping matrix, M is a mass matrix, u, u', u'' are displacement vectors, and F is a forces' vector. This formula is used to calculate the displacements and forces.

The programming statistics of SubSystemA are summarized in Table 1. These statistics were collected at revision *5c8366d* using the cloc utility program[1] (it was used to identify the programming languages used in SubSystemA, count the lines of code and the lines of comments) and the grep pattern matching utility program[2] (it was used to count the number of subroutines, functions, and Fortran modules).

[1] `http://cloc.sourceforge.net`

[2] `http://linuxcommand.org/man_pages/grep1.html`

Table 1: The statistics of SubSystemA

Statistic	Value
Languages used	Fortran 90
# Lines of code	76,332
# Lines of comments	25,051
# Subroutines	996
# Functions	31
# Modules	169

It appears that the major part of SubSystemA (and in fact, SubSystemA is one large standalone program) is implemented in the Fortran-90 language[3] using the procedural programming paradigm (see Figure 1). As shown in the figure, the "mainb" sub-program[4] contains the loop that iterates over the time series. The 2 main sub-programs, "static_analysis" and "dynamic_analysis" are called from within that loop. The sub-programs that might implement the engineering domain features are called from the "mainb", "static_analysis", and "dynamic_analysis" sub-programs and modify the state of global variables. It seems that subroutines particularly (due probably to their large number, see Table 1) and global variables in Fortran modules play a major role in the implementation of SubSystemA:

- Subroutines: these are the Fortran void-return-type sub-programs (functions are another type of sub-programs in Fortran that return values). In SubSystemA, these subroutines usually modify the state of global variables.
- Fortran modules: these are the "name-spaces" that contain the global variables and the global sub-programs.

According to Fortran-90 specification, the names of identifiers are no longer limited to 6 characters. Many identifiers in SubSystemA have meaningful long names. On average, for every 3 lines of code (executable statements), there appears to be 1 line of comment[5].

According to the estimation by the SubSystemA system's expert, there are:

- 43 engineering-domain features: these features implement the characteristics of the sub-sea structures and the environment such as the material properties, the presence of certain forces, the configuration of the structures, and others. These features augment/modify the K, D, M matrices in Formula 1.
- 12 analysis features: these features implement the FEA-based static and dynamic analysis in SubSystemA.

[3]There is a minuscule addition of C/C++ code

[4]In SubSystemA sub-programs are both subroutines and functions.

[5]In fact, the number of comments is higher: the cloc utility does not count the in-line comments

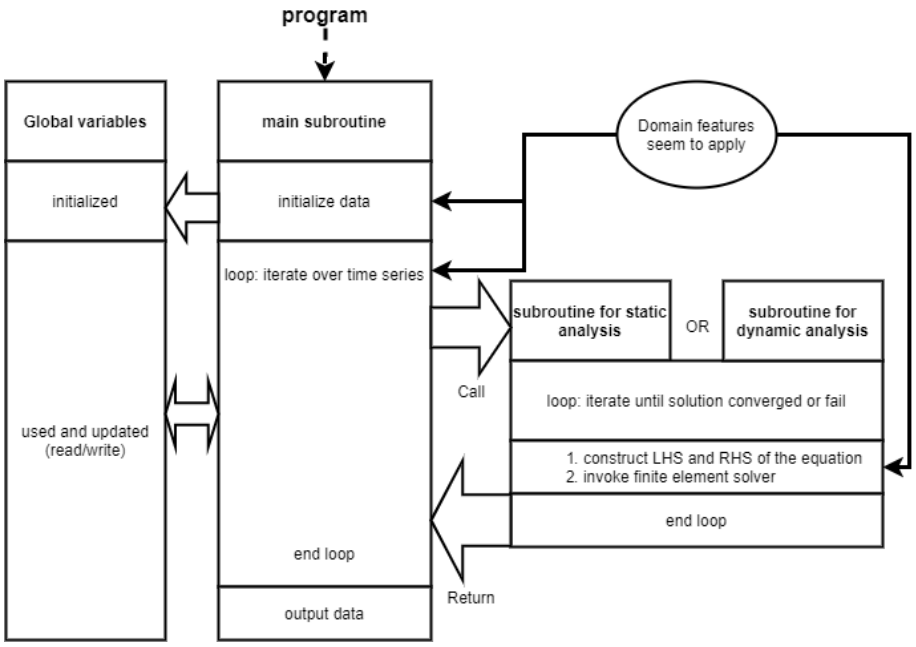

Figure 1: The abstract design of SubSystemA.

- 5 utility features: these features implement the input/output (I/O) in SubSystemA[6].

The *keyword file* signifies a parameter file to SubSystemA. It contains a set of parameters, organized in a structured way. Twenty nine of the names of the engineering domain features (as compiled by the system's expert) seem to correlate with the names of parameters in the keyword file. Therefore, it seems that at least some of the features in SubSystemA could be controlled via the keyword file by adding/modifying/removing the parameters.

Summarizing, there are several implications for feature location in SubSystemA. At least a part of the domain engineering features seem to be controlled via the keyword file. This means that these features potentially could be turned on and off and dynamic analysis could be employed. Because of the procedural programming paradigm, the features seem to be implemented using subprograms: the sub-program level of granularity could be used for FL. There seems to be a fair amount of textual information in source code: names of identifiers (the Fortran-90 standard has lifted the limitation for 6-character long names) and comments. Therefore, a textual analysis could be potentially used for feature location.

[6]Matrix operations are another type of utility features that could be assigned to this group.

4 Manual Feature Location in SubSystemA: Results and Analysis

To gauge how SubSystemA might respond to different source code analysis types it was decided to locate 4 features manually. We will refer to these 4 features are: "B", "G", "PIP", and "HL". These features were selected by the system expert to reflect a span of difficulty, from simple ("B") to difficult ("HL"). The following reasoning was used when selecting the type of analysis for this manual feature location:

- Dynamic source code analysis was selected, because a decent proportion of the features in SubSystemA seem to be controllable using the keyword file (see Section 3) and dynamic FLTs are good at identifying such controllable features (see Section 2.1).
- Textual source code analysis was selected, because SubSystemA seems to have a decent amount of meaningful textual data in its source code (see Section 3).
- Static code analysis was selected to be used in conjunction with the dynamic or textual analysis, after the starting points of a feature are found. (Because of the large number of source code components such as subprograms or modules, it doesn't appear to be efficient to use standalone static analysis, see Section 2.2).
- Because of the large number of change-sets (7793 at revision 5c8366d) and long history of recorded changes (almost 8 years) in SystemA's analysis part, the VCS could be used for historical source code analysis. It was decided to keep this analysis as an alternative, if some of the analysis above don't work with SubSystemA. (This type of analysis was tried with the "B" and "G" features, but wasn't used with the other two features, because the other types of analysis were sufficient to locate the features and because the historical analysis seemed to overlap with the static analysis (similar source code components were located).)

4.1 The Assessment of Manual Feature Location: The Gold Set and the Metrics

The output of FL is a set of source code components (for convenience, it is called simply the "result set"). To assess the effectiveness of a FL, for every given feature, one needs to compare the result set, with the set of known source code components for a given feature. The set of known features and known correct mappings to source code components for these features is called the "gold set". In this analysis, the gold set was identified in two steps: for each of the 4 selected features, the author of this report initially compiled the set of potentially relevant source code components, based on his own increasing

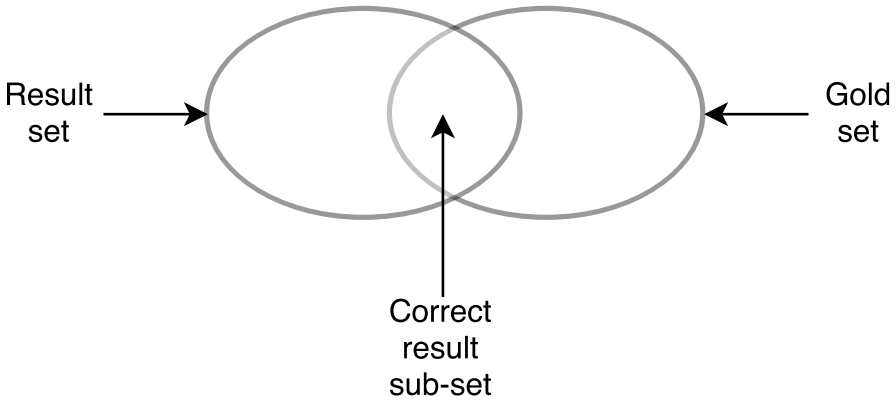

Figure 2: The correct result sub-set.

knowledge of the system, and the system expert later revised that set. The statistics for the gold set are shown in Table 2.

Table 2: The statistics of the gold set for the 4 features selected

Feature	Source code components			
	# Statements	# Conditional/Loop Blocks	# Modules (minus sub-programs)	# Sub-programs
B	1 (0.14%)	10 (20.30%)		2 (79.56%)
G	2 (2.53%)			1 (97.47%)
PIP	5 (0.16%)	31 (5.81%)	1 (3.76%)	24 (90.27%)
HL		3 (0.33%)		13 (99.67%)

The table shows the number of related source code components of sub-file-level granularity (features in SubSystemA are usually implemented across parts of several files, and hence a finer level of granularity is needed) for each of the 4 features. The numbers inside the parentheses indicate how much each type of source code components contributes to a feature's implementation in terms of lines of code. As can be seen, the sub-programs contribute the most, followed by the conditional/loop blocks. Hence, sub-programs were used as an approximation, when evaluating the result set, returned by manual FL.

Then, the set of correctly returned source code components (called the "correct result sub-set" in this work), identified by the manual FL approach for each feature, can be obtained by intersecting the gold set with the result set, as shown in Figure 2. Knowing these sets, a number of metrics could be applied to assess the effectiveness of a FL. For example, one could calculate the proportion of the correct result subset in the result set or the proportion of the correct result subset in the gold set: the former is called the *precision* metric and the latter the *recall* metric (see Equation 2 and Equation 3 respectively).

$$Precision = \frac{correct\_result\_subset}{result\_set} \tag{2}$$

$$Recall = \frac{correct\_result\_subset}{gold\_set} \tag{3}$$

.

There do not seem to exist any accepted standards of thresholds/ranges for precision and recall, but a technique showing recall and precision above 50% looks very reasonable. In this work, the focus is to maximize recall (for example, we could aim for a target recall of over 80% across test features), while keeping the adequate precision adequate. This is because the proposed approach is iterative where results are presented to an experienced developer. It is easier for these developers to look at a superset of the "correct result sub-set" and discard elements that are not appropriate than to see a subset and try to remember the elements that are not present.

These two metrics, however, do not take into account the ranked positions of relevant elements, which, for instance, could apply to textual source code analysis. This type of analysis usually returns lengthy lists of ranked results and the effectiveness in this case is measured by how many correct results are located in the top positions. Another metric, called the *average precision*, is more suitable in such cases. It is calculated for all correct results at every position k in a result set as shown in Equation 4.

$$AvgPrecision = \frac{\sum_k Precision(k)}{number\_of\_correct\_results} \tag{4}$$

For example, in case that returns 10 ranked answers, where the answers in position 1, 2, and 10 are known to be correct, the average precision is equal to $AvgPrecision = (1/1 + 2/2 + 3/10)/3 = 0.76$.

4.2 The Dynamic FL Method

For dynamic FL, the keyword files were used to create the execution scenarios. For every feature, at least one pair of keyword files was prepared, using the existing keyword files as a template, so that one keyword file in the pair triggers the execution of a feature and the other keyword file does not.

SubSystemA was instrumented to collect the execution traces by rebuilding it using the "Qcov-gen" compiler switch. For each pair of keyword files, the execution traces obtained when the feature was off were then subtracted from the execution traces obtained when the feature was on, using the differential coverage of the "codecov" utility[7]: only the source code components that are uniquely used when the feature is "on" are highlighted after this operation.

4.3 The Textual FL Method

To facilitate the textual FL, a small Java program was written. The program works as follows:

[7] `https://software.intel.com/en-us/node/680224`

1. It takes a search keyword as an input (see the Table 3 for the actual keywords used).

2. For each source code component of sub-program level granularity, a number of textual matches with the search keyword is calculated. (Any textual information within the body of a sub-program is used.)

3. The sub-programs are then ranked according to the number of matches with the search keyword.

Table 3: The keywords used for textual search.

Feature	Keywords
B	"buoyancy"
G	"gravity"
PIP	"pipe-in-pipe" "pip"
HL	"hydro" "drag" "added mass"

4.4 The Results of Dynamic and Textual FL

The results of dynamic FL are shown in Table 4. The results indicate that:

- As expected, dynamic analysis works only in the cases when it is possible to "turn on/off" the feature (2/4 cases): the other 2 features "B" and "HL" seem to be always executed. It is interesting to note here that even the presence of a corresponding keyword in the keyword file, was no guarantee that the feature could be turned off, suggesting the scope of this approach is more limited than previously thought.

- Because of this, the effectiveness of dynamic analysis is characterized by an average 48.22% precision and 38.55% recall. This means that only one third of sub-programs are identified on average and every second result is incorrect.

The results of the textual FL are summarized in the Table 5:

- It worked in all cases (4/4).

- Unsurprisingly, the textual FL is sensitive to the search query (see how the results differ when different search keywords are used in Table 5).

Table 4: The results of dynamic FL.

Feature	Precision %	Recall %
B	0	0
G	100	100
PIP	92.86	54.2
HL	0	0
Average	48.22	38.55

- The average of the average precision is 50.23%, which suggests that towards the beginning of the ranked list every second result is correct.
- The average recall of 66.6%, means that two thirds of a feature's sub-programs could be identified. In fact, the average recall decreased, because of the "HL" feature. The average recall for that feature was just 35.93%.

Table 5: The results of textual FL.

Feature	Keywords	Average Precision %	Recall %
B	buoyancy	32.5	100
G	gravity	100	100
PIP	pipe-in-pipe	44.33	66.7
	pip	65.06	91.67
HL	hydro	63.19	30.8
	drag	34.95	46.2
	added mass	11.55	30.8
Average		50.23	66.6

These results suggest that a textual FL approach might be more useful as it works in all cases and on the average achieves an average precision of 50.23% and an average recall of 66.6%. The textual approach, however, should be improved as high recall is important in the envisaged process. For example, the average recall for "HL" is just 35.93%, which means that only one third of the feature's sub-programs are located by this method alone.

Compounding the results above, the textual analysis locates source code components of just one level of granularity (sub-programs in this case) and misses the conditional/loop blocks that seem to be the second biggest group of source code components in a feature (see Table 2). These source code components should be located to cover as much of a feature's extent as possible.

4.5 An Illustrative Example: Improving the Recall of "HL" with Hybrid Textual-Static Approach.

One way to combat the low recall of textual FL applied to "HL" (and similar features) is to mix the textual analysis with static analysis. Particularly, the structural neighbourhood of a relevant sub-program, identified in the textual analysis, could be analysed to discover other related sub-programs (see Section 2.2). The intuition here is as follows: if the callees (sub-programs) of this sub-program have a small fan-in (a small number of distinct callers) then they are likely to belong to the same feature; similarly, if the callers (sub-programs) of this sub-program have a small fan-out (a number of distinct callees) they are also likely to belong to the same feature. To facilitate the static analysis, the Doxygen[8] tool was used (it generated the call graph for SubSystemA).

For example, the top 5 results of the textual search using the "hydro" keyword for the "HL" feature contain 3 related sub-programs in positions 1, 3, and 4. By inspecting the structural neighbourhood of these 3 sub-programs in line with the guidelines above, another 7 relevant sub-programs were located[9]. The recall of this approach was 76.92% (10 of 13 sub-programs found), whereas the recall for the textual only FL was 30.8% (3 of 13 sub-programs found).

4.6 An Illustrative Example: Finding the Conditional/Loop Code Blocks with a Different Hybrid Textual-Static Approach

Some features might have a fairly high percentage of their implementation allocated in conditional/loop blocks of code. A good example is the "B" feature at 20.3% (see Table 2). Observation of the SubSystemA code in this instance suggests that variables control access to the code blocks and these code blocks call relevant sub-programs: $variable \rightarrow block \rightarrow sub-program$. The assumption in this case is that the same variables may control other feature-relevant blocks.

For example, the top 5 results of the textual search using the "buoyancy" keyword for the "B" feature contain 2 related sub-programs in positions 4 and 5. The first is called from 2 code blocks that are controlled by two variables. These 2 variables appear to control 8 other code blocks (10 total). Of these 10 code blocks, 9 belong to the "B" feature (90% of the relevant code blocks are thus identified: a recall of 90% with respect to blocks of code and 100% with respect to sub-programs).

5 The Proposed Technique

The FL approach, as described in the previous section, would require a lot of effort from a developer if done manually. He/she has to identify a tiny amount of

[8] `http://www.stack.nl/~dimitri/doxygen/`

[9] The arbitrary threshold of 4 was used for fan-in and fan-out when selecting the relevant adjacent sub-programs.

relevant source code components in a much bigger pool of irrelevant components, to analyse structural relationships between components, and to remember and update the list of discovered relevant components. Fortunately, such process can be automated to a large extent.

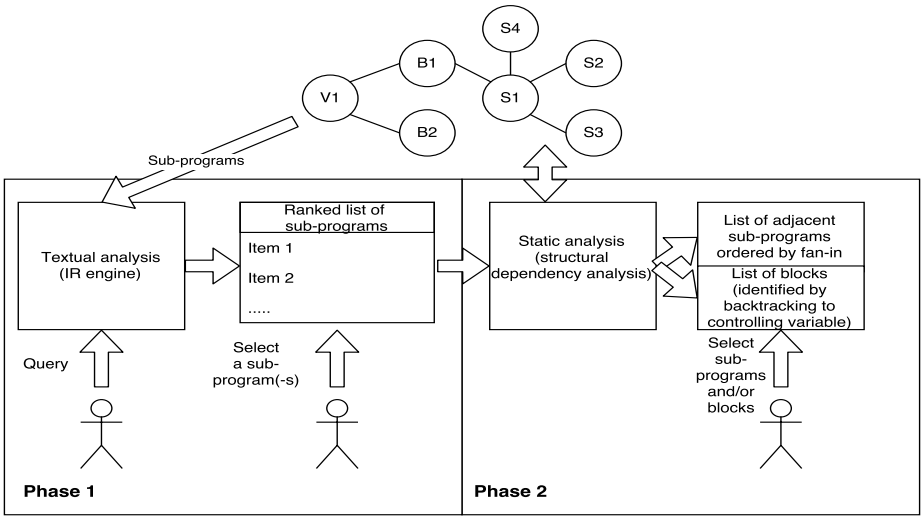

Figure 3: The proposed FLT.

Based on the characteristics of SubSystemA and the results of our manual feature location exercise, this section proposes, a semi-automated, two-phase, two-iteration hybrid FLT. An abstract representation of this technique is shown in Figure 3. This FLT is semi-/automated, because it still requires a human interaction. This interaction, however, is minimized and assisted (for example, the results presented to the user are ranked or ordered). The FLT is separated into two-phases: the first phase utilizes textual analysis to find initial locations where a feature is implemented (at sub-programs level) and the second phase uses those starting locations to identify other relevant source code components (sub-programs and blocks). The second phase of the FLT allows for two iterations, which means that once the source code components, that are structural dependencies, were discovered the approach could be repeated once for these newly discovered components: for example, if one identifies a sub-program called by a relevant sub-program from step 1, he/she may iterate to its "children".

Below the steps of this approach are outlined in more detail:

1. *Build a structural dependency graph (SDG).* In the first step, the technique parses the source code and constructs the SDG. The parser is an initial step in the analysis, used to transform the source code into a data structure (SDG in our case), that can be used to determine the structural dependencies between code components in our analysis. The following source

code components are parsed: variables that appear in conditional/loop blocks (V), conditional/loop blocks (B), sub-programs (both subroutines and functions) (S). The V and S have connections to B, but not to each other; the B has connections to both V and S; the S has connections to other S. These components (nodes in the SDG) and their relationships are shown in Figure 3. In the Figure, V1 controls access to blocks B1 and B2, B1 invokes S1, and S1 invokes S2, S3, and S4. The SDG has to be updated, to mark the nodes that are selected as part of a feature in phase 1 and phase 2, so that they do not appear/affect the search results. However, fan-in/fan-out (or any other statistic that will be used in phase 2 should remain intact)[10].

2. *Phase 1: Textual analysis.*

   (a) *Prepare sub-programs for textual analysis via the IR.* In the next step, the S of SDG are prepared for textual analysis via the IR, using the steps described in Section 2.3. The textual data for each $s \in S$ is extracted and passed as individual documents to the IR engine, implementing the VSM IR model[11].

   (b) *Search for the starting points (sub-programs).* The first user interaction requires him/her to submit a search query to the IR engine. This search query has to somehow describe a feature (for example, the keywords used in the textual analysis in the previous section). The result of this operation is a ranked list of sub-programs, returned by the IR engine to the user.

   (c) *Inspect the search results.* The user inspects the ranked list and marks the sub-programs that he/she deems to be relevant to a feature. At this point, it could happen that the feature extent has been fully identified. For example, the feature could consist of just one sub-program (e.g. the "G" feature). In this case the relevant sub-programs and the statements that call these sub-programs are identified as the implementation of a feature[12]. A user then may proceed with the next feature. More typically though, a user may decide that the feature has not been fully identified at this point and that further analysis is needed, in which case they proceed to the next step (phase 2).

3. *Phase 2: Static analysis.* If a user has decided to proceed to phase 2, he/she may choose one or both of the structural analysis types: find ad-

[10]However, when the code changes (for example, a user checks a new revision of source code), the SDG and the fan-in/fan-out should be updated to reflect the new structure of code

[11]Currently there are mixed opinions in FL literature as which IR model to choose: VSM models seem to be simple and effective, whereas the more sophisticated LSI and LDA haven't shown to be significantly more effective than VSM.

[12]After sub-programs are marked as belonging to a feature, the IR index should not be updated because it is computationally expensive. Simple filtering of search results (removing sub-programs that are already part of a feature) will work as an approximation. However, the IR index should be updated if the source code changes (for example, a user checks a new revision)

jacent sub-programs and/or find related code blocks. The sub-programs that a user selects in the first phase from the ranked list, are the input to the static analysis in this phase.

(a) *Find adjacent sub-programs.* This type of analysis allows the user to inspect the other sub-programs, that are direct neighbours of the input sub-programs. To further assist a user, these adjacent sub-programs are arranged in such an order that the sub-programs with the smallest fan-in/fan-out appear at the top of the list. For example, in Figure 3, the sub-program $S1$ has 3 adjacent sub-programs that would be ordered according to their fan-in/fan-out statistic if $S1$ is an input sub-program from phase 1. A user inspects this list of sub-programs and marks those related to the feature: these sub-programs and their calling statements become a part of a feature. This step allows for two iterations, meaning that the neighbourhood of the sub-programs in the list may be further explored[13].

(b) *Find related code blocks.* This type of analysis allows the user to inspect the code blocks, that could be a part of the same feature of a specified sub-program. This is done by following the relationship from the sub-program to the code block that calls that sub-program and to the variable(-s) that controls that block. Other code blocks, connected to the variable(-s), are shown to a user so that he/she can decide if these code blocks belong to the feature. For example, in Figure 3, the sub-program $S1$ is called from the block $B1$, which is controlled by the variable $V1$. $V1$ also controls the block $B2$, which might also be relevant to a feature.

The output of the technique (after phase 1 and phase 2) is a list of source code components (sub-programs, conditional/loop blocks, call statements) linking to the code.

Two optional optimizations to this approach could be:

- Filtering of sub-programs in the ranked list of search results in phase1. It was noticed that there exist large subroutines in SubSystemA with big fan-out of 30 and more (e.g. "mainb" subroutine). These subroutines do not seem to be a part of any domain engineering feature, but appear in the top positions because of the high number of textual matches. Removing these subroutines from the search results could improve the average precision of the textual analysis. Yet, these subroutines seem to be a part of FEA-based analysis features. A reasonable solution could be to optionally add/remove them from the ranked list depending on what type of a feature a developer is locating.

[13]The reason for this is because during the manual feature location only the directly connected sub-programs were found to be relevant. Going deeper into the call graph of sub-programs didn't reveal any additional relevant sub-programs. But it was tried with the "HL" feature only. Hence, two iterations seem more appropriate.

- Inspect sub-programs, called from the conditional/loop blocks, identified in phase 2. If the code block belongs to a feature then the sub-programs called from this block could also belong to a feature. This could potentially further improve the recall of the approach.

These optional optimizations, are "stretch" improvements and will be considered if there is time for these improvements (there are certain risks associated with optimizations such as overcomplicating the technique and failure to meet the main goals). The second optimization could potentially improve the recall (which is our priority) and will be considered first. The first optimization could improve the average precision and will be considered second.

6 Conclusions

Feature location in SubSystemA is the first step of the analysis engine source code unification project. According to Dit et al., there are several types of source code analysis targeted at feature location: dynamic, static, textual, historical, and their combinations [10].

Several characteristics of SubSystemA were identified that could affect the selection of the FLT: some features in SubSystemA seem to be controlled via the keyword file; SubSystemA is implemented using procedural programming, where subroutines and global variables play a main role; there is a decent amount of meaningful textual information in source code such as names of the identifiers and comments.

To gauge how SubSystemA would respond to different types of source code analysis, 4 engineering domain features were located manually. Textual analysis was able to locate feature-relevant sub-programs with an average precision of 50.23% and recall of 66.6% on the average. Two illustrative examples showed that this could be further improved by using static analysis of structural code dependencies: for "HL" recall was improved and, likewise, for "B" additional conditional/loop code blocks were found.

As a result of these findings and observations, a semi-automated two-phase two-iteration hybrid (textual-static) approach was proposed. The approach could be further optimized by filtering the ranked results from the textual analysis phase and suggesting the sub-programs called from relevant code blocks.

The major benefit of this approach is in reducing the developer's effort by automating monotonous repetitive and complex tasks. The more detailed benefits of this approach are:

- The proposed approach seems to be widely applicable, unlike a dynamic analysis approach.
- It combines the state-of-the-art technique for finding a foothold into a feature and the state-of-the-art technique for full feature recovery.
- It is also customized to the characteristics of the SubSystemA system to include conditional/loop blocks.

- The approach demonstrates high recall and high/adequate average precision over the trials we have carried out: the mix of metrics desirable in this approach.

The limitations of this approach are really to be determined in further trials. Currently, the limitations seem to arise from the initial trial:

- The approach demonstrates a high recall, however, some source code components may still have to be located manually.
- The trial studied 4 features only and may not be entirely valid across unseen features.
- It was tried with 1 system only (SubSystemA) and so may be biased to work best with SubSystemA.

References

[1] Lauren R. Biggers, Cecylia Bocovich, Riley Capshaw, Brian P. Eddy, Letha H. Etzkorn, and Nicholas a. Kraft. Configuring latent Dirichlet allocation based feature location. *Empirical Software Engineering*, 19(3):465–500, aug 2014.

[2] DM Blei, AY Ng, and MI Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[3] Frederick P Brooks. *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering*. Pearson Education, 1995.

[4] Jim Buckley, Sean Mooney, Jacek Rosik, and Nour Ali. Jittac: a just-in-time tool for architectural consistency. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1291–1294. IEEE, 2013.

[5] Gerardo Canfora and A. Cimitile. *Handbook of Software Engineering & Knowledge Engineering: Fundamentals*. River Edge NJ: World Scientific, 2001.

[6] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Comput. Soc, 2000.

[7] Muslim Chochlov, Michael English, and Jim Buckley. A historical, textual analysis approach to feature location. *Information and Software Technology*, 88:110–126, 2017.

[8] Scott Deerwester, ST Dumais, and TK Landauer. Indexing by latent semantic analysis. *Journal of the American Society for Information Science (1986-1998)*, 41(6):391–407, 1990.

[9] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can Better Identifier Splitting Techniques Help Feature Location? In *IEEE 19th International Conference on Program Comprehension*, pages 11–20. Ieee, jun 2011.

[10] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code : a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[11] Abbas Heydarnoori, Krzysztof Czarnecki, Walter Binder, and Thiago Tonelli Bartolomei. Two Studies of Framework-Usage Templates Extracted from Dynamic Traces. *IEEE Transactions on Software Engineering*, 38(6):1464–1487, 2012.

[12] Tara Kelly and Jim Buckley. A context-aware analysis scheme for bloom's taxonomy. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 275–284. IEEE, 2006.

[13] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek , Relate , and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering (2006)*, 32(12):971–987, 2006.

[14] Walid Maalej and Hans-J{\"o}rg Happel. From Work to Word : How Do Software Developers Describe Their Work ? In *Mining Software Repositories (MSR), 2009 6th IEEE Working Conference on*, pages 121–130, 2009.

[15] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*, pages 214–223. IEEE Comput. Soc, 2004.

[16] Michael P O'Brien, Jim Buckley, and Christopher Exton. Empirically studying software practitioners-bridging the gap between theory and practice. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 433–442. IEEE, 2005.

[17] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. How to Effectively Use Topic Models for Software Engineering Tasks ? An Approach Based on Genetic Algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 522–531, 2013.

[18] Maksym Petrenko and Václav Rajlich. Concept location using program dependencies and information retrieval (DepIR). *Information and Software Technology*, 55(4):651–659, apr 2013.

[19] Sukanya Ratanotayanon, Hye Jung Choi, and Susan Elliott Sim. Using transitive changesets to support feature location. In *Proceedings of the*

IEEE/ACM international conference on Automated software engineering - ASE '10, pages 341–344, 2010.

[20] Martin P Robillard. Topology Analysis of Software Dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):18:1—-18:36, 2008.

[21] Gerard Salton and Michael J McGill. *Introduction to modern information retrieval*. 1983.

[22] Giuseppe Scanniello and Andrian Marcus. Clustering Support for Static Concept Location in Source Code. In *IEEE 19th International Conference on Program Comprehension*, pages 1–10. Ieee, jun 2011.

[23] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An Examination of Software Engineering Work Practices. In *CASCON '10 CASCON First Decade High Impact Papers*, pages 174—-188, 2010.

[24] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. An Exploratory Study of Feature Location Process. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 213–222, 2011.

[25] Norman Wilde and C. Michael Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[26] Anis Yousefi and Kamran Sartipi. Identifying distributed features in SOA by mining dynamic call trees. In *27th IEEE International Conference on Software Maintenance*, pages 73–82, 2011.

[27] Celal Ziftci and Ingolf Kruger. Feature Location Using Data Mining on Existing Test-Cases. In *19th Working Conference on Reverse Engineering*, pages 155–164. Ieee, oct 2012.