Title: Improving Feature Location Using Structural Similarity and Iterative Graph Mapping

Authors: Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, Wenyun Zhao

PII: S0164-1212(12)00300-7
DOI: doi:10.1016/j.jss.2012.10.270
Reference: JSS 9041

To appear in:

Received date: 15-2-2012
Revised date: 25-10-2012
Accepted date: 25-10-2012

Please cite this article as: Peng, X., Xing, Z., Tan, X., Yu, Y., Zhao, W., Improving Feature Location Using Structural Similarity and Iterative Graph Mapping, *The Journal of Systems and Software* (2010), doi:10.1016/j.jss.2012.10.270

Improving Feature Location Using Structural Similarity and Iterative Graph Mapping

Xin Peng[1], Zhenchang Xing[2], Xi Tan[1], Yijun Yu[3], Wenyun Zhao[1]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]Department of Computing, The Open University, UK
[1]{pengxin, 09210240034, wyzhao}@fudan.edu.cn, [2]zcxing@ntu.edu.sg, [3]y.yu@open.ac.uk

Abstract. Locating program element(s) relevant to a particular feature is an important step in efficient maintenance of a software system. The existing feature location techniques analyze each feature independently and perform a one-time analysis after being provided an initial input. As a result, these techniques are sensitive to the quality of the input. In this paper, we propose to address the above issues in feature location using an iterative context-aware approach. The underlying intuition is that features are not independent of each other, and the structure of source code resembles the structure of features. The distinguishing characteristics of the proposed approach are: (1) it takes into account the structural similarity between a feature and a program element to determine feature-element relevance; (2) it employs an iterative process to propagate the relevance of the established mappings between a feature and a program element to the neighboring features and program elements. We evaluate our approach using two different systems, DirectBank, a small-scale industry financial system, and Linux kernel, a large-scale open-source operating system. Our evaluation suggests that the proposed approach is more robust and can significantly increase the recall of feature location with only a minor decrease of precision.
Keywords: feature location, traceability recovery, information retrieval, structural similarity

1. Introduction

Software maintenance tasks, such as bug fixes and feature enhancements, often originate from the users of a system. The description of these tasks is often expressed in terms of features that are observable to the users. Locating program element(s) relevant to a particular feature (i.e., feature location) is an important and recurring step in efficient maintenance of a software system. Researchers have presented techniques to provide automated assistance in feature location. In particular, researchers have investigated using Information Retrieval (IR) (i.e., lexical analysis) (Marcus and Maletic, 2003; Poshyvanyk and Marcus, 2007; Zhao et al., 2006; Marcus et al., 2004), static analysis (Warr and Robillard, 2007), dynamic analysis (Eisenbarth et al., 2003), and the hybrid of several analysis techniques (Poshyvanyk and Gueheneuc, 2007).

These existing techniques analyze each feature independently, ignoring the interdependencies (i.e., structural context) of features and the interdependencies of program elements. Furthermore, they perform a one-time analysis after being provided an initial input. As a result, these techniques are sensitive to the quality of the input, for example, the quality of feature descriptions, the quality of the identifiers and comments of program elements, or the availability of carefully designed test cases.

To address the above issues, we propose an Iterative Context-aware approach to automatic Feature Location (ICFL). Our ICFL approach is inspired by two lines of research. First, research on domain engineering (Kang et al., 1990; Zhang et al., 2006) suggests that features are not independent of each other, for example, a feature may use, extend, or refine other features; features and their interdependencies form some forms of feature models (Peng et al., 2006). Furthermore, using appropriate reverse-engineering techniques (such as She et al., 2011; Dumitru et al., 2011; Yang et al., 2009), we can construct such feature models from existing requirement documents and source code with the help of domain experts. Second, research on distributed information retrieval (Ermolayev et al. 2005) suggests that similar elements in two different knowledge models usually have similar structural contexts. These two lines of research suggest that in addition to the "local" properties (such as lexical descriptions) of features and program elements the structural context of features and program elements can serve as the other important factor for determining the relevance between features and program elements.

Based on the above observations, our ICFL approach takes as input a requirement model that captures features and their interdependencies and a program model that captures program elements and their interdependencies. It solves feature location problem by computing many-to-many feature-element mappings between the two input models. ICFL measures the relevance between a feature and a set of

program elements based on their lexical and structural similarity (Section 3.3). The lexical similarity measures the similarity of feature description and source code (e.g. identifiers and comments). The structural similarity measures how much of the structural context of a feature can be mapped to the structural context of a set of program elements based on already-established feature-element mappings.

ICFL uses a greedy best-first search algorithm to incrementally recover feature-element mappings. It employs an iterative process to propagate the knowledge of the established mappings between a feature and a set of program elements to the neighbouring features and program elements (Section 3.4). The underlying intuition is that the more feature-element mappings ICFL recovers, the more likely it becomes that ICFL may recover further related feature-element mappings. To avoid the negative impact of erroneously feature-element mappings on the subsequent mapping iterations, ICFL starts with a high initial similarity threshold and gradually decreases the similarity threshold in the subsequent iterations. This strategy ensures that feature-element mappings with higher similarities would be identified first and then these established mappings would help to identify more feature-element mappings in the subsequent iterations.

We evaluate our ICFL approach using two subject systems, DirectBank system and Linux kernel. The DirectBank system is a small-scale Java-based financial system from our industry partner, Wingsoft Ltd., which provides bank interfacing services for cashiers. DirectBank system consists of 30K lines of code, 71 features and 951 feature interdependencies, 53 classes and 414 methods. The Linux kernel is an open source operating system kernel written in C; it consists of about 12 million lines of code, 2000 directories and 28,024 files, and 155,266 C functions. Furthermore, Linux kernel contains about 700 Kconfig (a Linux-kernel specific feature modelling language (Berger et al., 2010; She et al., 2010)) files, from which we automatically extract the feature model of Linux kernel (consisting of 6,052 features and 1,410,830 interdependencies). We compare our ICFL approach with an IR-based approach to feature location (Zhao et al., 2006). Our evaluation suggests that the proposed iterative context-aware approach to feature location is more robust and can significantly increase the recall of feature-element mappings with only a slight decrease in the precision.

The remainder of the paper is organized as follows. Section 2 introduces related work. Section 3 describes the proposed ICFL approach, including an overview of the ICFL algorithm, meta-model, similarity metrics, the details of the ICFL algorithm and the output. Section 4 presents the evaluation of our approach with DirectBank and Linux kernel systems. Section 5 discusses several relevant issues and threats to validity of our ICFL approach. Section 6 concludes and outlines future work.

2. Related Work

A large number of techniques for feature location have been proposed. They can be divided into (a) IR-based approaches, such as Latent Semantic Indexing (LSI) (Marcus and Maletic, 2003; Poshyvanyk and Marcus, 2007; Marcus et al., 2004), (b) dynamic approaches that analyze execution scenarios (Eisenbarth et al., 2003), and (c) hybrid approaches that combine information retrieval, static and/or dynamic analysis (Poshyvanyk and Gueheneuc, 2007). Dit et al. (Dit et al., 2011) provides a comprehensive survey of existing feature location techniques.

Among IR-based feature location approaches, Latent Semantic Indexing (LSI) is frequently used to capture the essential semantic information of each feature by dimension reduction (Marcus et al., 2004). Poshyvanyk and Marcus (Poshyvanyk and Marcus, 2007) propose to combine LSI-based analysis and Formal Concept Analysis (FCA). FCA is employed to analyze the relations between program elements and features (called objects and attributes in FCA respectively). Dynamic-analysis based feature location approaches, such as (Eisenbarth et al., 2003), examine traces collected from a set of scenarios of system executions; some of these scenarios execute features, while the others do not. A recent work by Kazato et al. (Kazato et al., 2012) presents a feature location approach for multi-layer systems, which merges execution traces collected in each layer using dynamic analysis and then uses FCA to identify collaborative program elements for each feature. Poshyvanyk and Gueheneuc (Poshyvanyk and Gueheneuc, 2007) propose a hybrid approach to combine dynamic analysis (by scenario-based probabilistic ranking of events) and information retrieval (LSI-based technique) to overcome the uncertainty in decision makings of feature location.

These feature location approaches analyze each feature independently and perform a one-time analysis. For example, in IR-based feature location approaches features are considered as separate concepts with different descriptions; and in dynamic-analysis based approaches, features are only connected to relevant execution scenarios. The mappings between features and program elements are established by examining only such "local" properties. Adams et al. (Adams et al., 2010) presents COMMIT, a history-based technique for mining crosscutting concerns (features). This technique identifies weighted relations between

sub-concerns and thus reconstructs composite concern as multiple interconnected sub-concerns. In contrast, our ICFL approach takes into account the interdependencies (i.e., structural context) of features and program elements and performs an iterative process for determining the program elements' relevance to the features. Our position paper (Peng et al., 2011) introduces basic concepts of ICFL, while in this paper we illustrate our ICFL approach with more examples and evaluate ICFL with a large-scale empirical study using Linux kernel.

Zhao et al. (Zhao et al., 2006) present a two-phase approach for feature location called SNIAFL. SNIAFL first applies an IR technique to identify an initial set of feature-element mappings based on the lexical description of features and program elements; it considers features as individual requirements units with lexical descriptions. Then, SNIAFL enriches the initial mappings by exploring only program call graph. In a recent work, McMillan et al. (McMillan et al., 2011) presents a code search system called Portfolio. It combines information retrieval and call-graph analysis for feature location. Similar to SNIAFL, Portfolio first finds initial focus points (relevant functions) by matching source code and keywords from queries. Then it uses PageRank and Spreading Activation Network (SAN) to rank the relevance of focus points. These two approaches consider only code-level structural information but not feature-level structural information. The quality of the initial feature-element mappings greatly affects the quality of the final results. Furthermore, SNIAFL uses code-level structural information in a separate phase to "improve" (e.g. extend or rank) the initial IR-based search results. In contrast, our ICFL approach considers the structural context of both features and program elements and measures the overall similarity between a feature and a program element based on both their lexical description and structural context at the same time. Furthermore, the iterative matching process of our approach allows ICFL to identify feature-element mappings with high similarities initially, and then identify more and more feature-element mappings in the subsequent iterations.

Lucia et al. (De Lucia et al., 2008) propose an incremental IR-based feature location approach and conduct a comparative study of one-shot and incremental approaches. However, this approach analyzes only the lexical descriptions and does not consider the structural context in its iterative feature location process. Their incremental process is guided by the incrementally decreased similarity threshold. This gives users the control over the number of validated correct links that they are interested in. This incremental process usually causes a significant decrease in precision with the increase in recall as the similarity threshold decreases, while our approach can significantly increase the recall of feature location with only a slight decrease of precision due to the consideration of structural context of features and program elements in the iterative matching process.

3. The Approach

In this section, we first give an overview of our ICFL approach. We then describe the meta-model assumed by our ICFL approach as the underlying representation for capturing the structural context (i.e., interdependencies) of features and program elements. Next, we discuss the similarity metrics on which ICFL relies to determine a program element's relevance to a feature. Finally, we present the iterative feature location algorithm and the output.

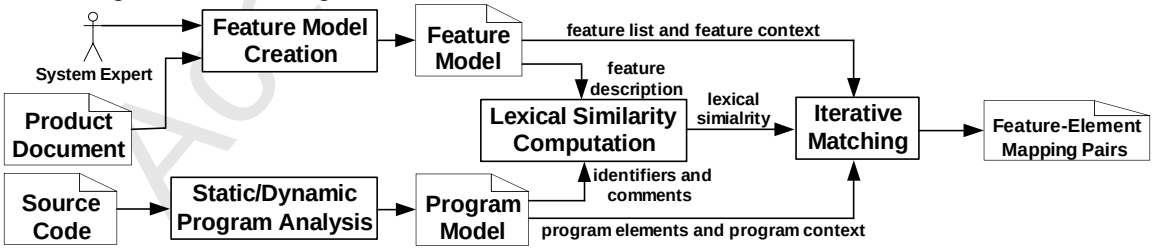

Figure 1. Overview of the ICFL approach

3.1 Overview

Figure 1 presents an overview of our ICFL approach. The program model can be easily extracted using static or dynamic program analysis techniques, which capture the lexical description (e.g. identifiers and comments) and the structural information of program elements. The feature model is usually created by system experts. For example, software product lines such as Linux kernel are equipped with feature models for feature management and product derivation (Berger et al., 2010). In the absence of such forward-engineered feature models, feature models can be reverse-engineered from existing product documents and

source code (She et al., 2011; Dumitru et al., 2011; Yang et al., 2009). Feature models capture feature descriptions and the interdependencies between features.

Lexical similarity can be computed between feature description and program description (i.e., identifiers and comments) using IR techniques. Furthermore, structural feature relations and program relations are extracted from feature model and program model as feature and program contexts, respectively. Then based on those already established feature-element mapping pairs, the structural similarity between a feature *f* and a program element *p* can be computed by evaluating how much of *f*'s neighbouring features and their relations can be mapped to *p*'s neighbouring program elements and their relations according to predefined mapping schema between feature relation types and program relation types.

Based on structural and lexical similarities between features and program elements, ICFL uses a greedy best-first search algorithm to incrementally recover feature-element mapping pairs. It employs an iterative process with several rounds of mapping. The initial (first) round mapping is based on lexical similarity only, and in following rounds lexical similarity and structural similarity are combined for determining the correspondences between features and program elements. In each round, if some candidate feature-element mapping pairs are identified, then the structural contexts of these newly mapped feature-program-elements pairs are activated, and the similarities of related not-yet mapped candidate pairs will be re-computed. As a result, more mapping pairs may be accepted due to the similarity re-computation in the next round.

In the first few iterations, lexical similarity is often the main factor for determining the correspondences between features and program elements. However, in the following rounds of iterative mapping, as more and more structural contexts of already-established feature-elements mappings are activated, structural similarity plays an increasingly significant role in determining the mappings between features and program elements. To minimize the negative impact of erroneous feature-element mapping pairs on the following mapping iterations, ICFL adopts the strategy of a high initial threshold with gradual concession in subsequent iterations. This strategy ensures that feature-element mappings (especially those identified in initial iterations of ICFL) are most likely correct.

3.2 Meta-model of the Feature and Program Models

Our ICFL approach solves the feature location problem by computing many-to-many feature-element mappings between the two input models, i.e., a feature model representing features and their interdependencies as well as a program model representing program elements and their interdependencies, respectively. The meta-model of the input models that ICFL assumes is a typed directed graph.

In a typed directed graph, each graph *node* represents one individual element of the model, associated with a node *type*. In this study, a feature model consists of two types of graph nodes, i.e., use-observable *functional features* and *business objects*. A *business object* denotes a user-observable business entity that can be produced, modified or accessed by *functional features*. A program model consists of three types of graph nodes, i.e., *classes/files*, *methods*, and *database tables* (*db_tables*). For object-oriented programs such as the DirectBank system, *methods* are regarded as implementation elements for *functional features*, *classes* denote entity classes corresponding to *business objects*, and *database tables* represent persistent objects. For procedural programs such as Linux kernel, *files* are regarded as implementation elements for *functional features*. Each node in a feature model or a program model has an attribute *description*, such as the natural language description of the feature, the identifiers and comments of the method.

Edges of a typed directed graph represent directed dependencies between model elements. Each edge is associated with an edge *type*. In this study, a feature model consists of two types of edges, representing *decomposition* between features and *read/update* dependencies between features and business objects, respectively; a program model captures *call* relations between methods and *data_read/data_update* dependencies between methods and classes/db_tables.

It should be noted that node and edge types that can be used in the ICFL approach are not limited to those used in the studies reported in this paper. More node and edge types can be defined based on application-specific knowledge about requirements and implementation. For example, for a web-based system, one can define web pages as a new node type and page links as a new edge type in the program model.

Because feature model and program model describe the system at different levels of abstraction, one may need to determine the correspondence between edge types in the two input models, in order to enable the computation of structural similarity. In ICFL, this correspondence is defined by users as an application-specific mapping schema between edge types of feature models and program models.

For example, in our empirical study with the subject system DirectBank (see Section 4.1), we define that *feature decomposition* corresponds to *method call*. We also define that *read/update* dependencies between features and business objects in feature models correspond to *data_read/data_write* dependencies between *methods* and *class*/*db_tables* in program models. This is based on the nature and implementation of the DirectBank system. That is, DirectBank is a database-centric information system (Yang et al., 2009). The business logics of a program element can be largely characterized by how this element operates on business data. Furthermore, our industrial partner informed us that in DirectBank *read/update* dependencies between features and business objects are implemented by data access (*data_read/data_write*) relations between *methods* and *class*/*db_tables*.

Figure 2 and Figure 3 present (partially) the feature model and the program model from our empirical study with DirectBank. In DirectBank, the functional feature *SignIn* is *decomposed* into two functional features *BankOpAuth* and *IPAddrAuth*, which deal with operator- and IP-based authorization respectively. Furthermore, the feature *SignIn reads* operators from the business object *OperatorList* and it *updates* another business object *OperationHistory* to log operation history. The feature *SignIn* is implemented in the method *Operator.login*, which *calls* two other methods *Operator.hasBankAuth* and *Operator.checkIp-Address* to check whether the operator and the client IP address are authorized. The method *Operator.login reads* the operators from the class *CurOperators* and logs the operation history in the database table *OperationLog*. It can be seen that the structural context of the features and the relevant program elements are similar, and this structural similarity would help to determine the relevance between features and program elements, even though their lexical descriptions are not similar (for example, the feature *SignIn* and the method *Operator.login*).

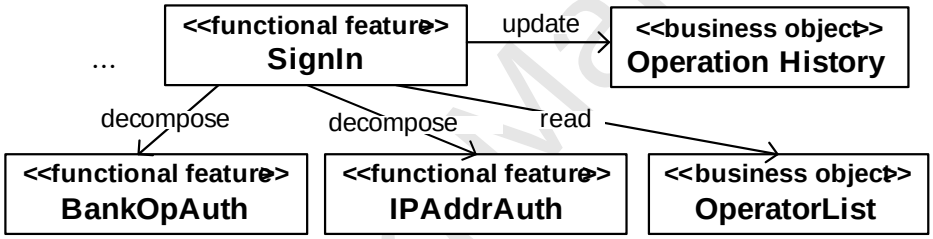

Figure 2. A partial feature model of DirectBank

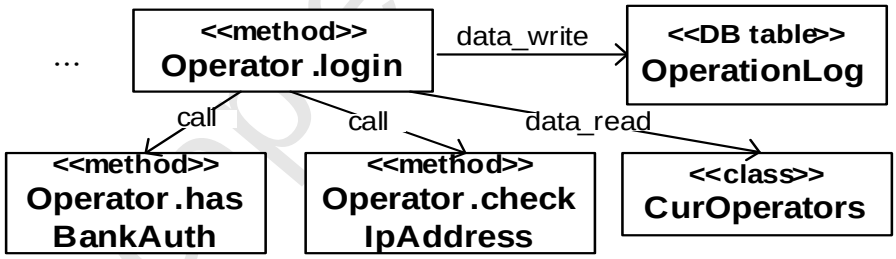

Figure 3. A partial program model of DirectBank

3.3 Similarity Metrics

Let us discuss similarity metrics for mapping a feature to the relevant program elements. The key to determine such mappings in ICFL is to compare the similarity between a feature and a program element, both at the lexical and at the structural level. In this subsection, we first introduce the lexical similarity, structural similarity, and the overall similarity metrics that ICFL uses. Then, we illustrate the computation of these similarity metrics with an example.

3.3.1 Lexical similarity

ICFL encodes the *description* attribute of a graph node (i.e., model element) in a Term-Frequency/Inverse-Document-Frequency (TF/IDF) vector (Baeza-Yates and Ribeiro-Neto, 1999). Recall that the *description* attribute of a graph node captures the lexical description of a corresponding feature or program element. ICFL constructs TF/IDF vectors for features based on feature descriptions and TF/IDF vectors for program elements based on their source code respectively. The TF/IDF vector evaluates how important a word is to a document in a corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. The TF/IDF

vector is often used in IR-based feature location approaches (Zhao et al., 2006) for determining a program element's relevance to a feature. In the similar vein, we define the lexical similarity between a feature and a program element as follows:

Definition 1 (Lexical Similarity)
Given a feature *f* and a program element *p*, ICFL measures their lexical similarity Sim_L with the cosine similarity between the TF/IDF vectors V_f and V_p, as defined in the following equation:

$$Sim_L(f,p) = \sum_{i=1}^{n} v_f[i]v_p[i] / \left(\sqrt{\sum_{i=1}^{n} v_f[i]^2} \sqrt{\sum_{i=1}^{n} v_p[i]^2} \right)$$

3.3.2 Structural similarity

ICFL employs an iterative feature location algorithm, involving multiple iterations of mapping between features and program elements (see Section 3.4). In each iteration (except the first iteration), when determining the relevance of candidate feature-element mappings, feature-element mappings established in earlier iterations provide the structural context for computing structural similarity of candidate feature-element mappings.

This structural context is captured by the neighbouring features (or program elements) as defined below.

Definition 2 (Neighbouring Features/Elements)
Given a feature (or program element) node *n* and an edge type *t*, let *neighbour*(*n*, *t*) be a set of neighbouring features (or program elements) consisting of all feature (or program-element) nodes that are related to *n* through edges of type *t*.

For example, according to the feature model and program model given in Figure 2 and Figure 3, there are:

neighbour(*SignIn*, *decompose*)={*BankOpAuth*, *IPAddrAuth*},

neighbour(*Operator.login*, *call*)={*Operator.hasBankAuth*, *Operator.checkIpAddress*}.

Furthermore, the computation of structural similarity depends on the mapping schema between the edge types of feature model and the edge types of program model, as defined below.

Definition 3 (Mappings of Edge Types between Feature Model and Program Model).
Given an edge type *ft* of feature model, let *map*(*ft*) be a set of the corresponding edge type(s) of program model, according to the user-defined application-specific edge-type mapping schema.

For example, in the example given in Figure 2 and Figure 3, we have *map*(*decompose*)={*call*}, *map*(*read*)={*db_read*}, and *map*(*update*)={*db_write*}.

Based on the above definitions, we now define a structural-similarity metric. In the *i*th iteration of the iterative mapping process, let M_i be the set of already established feature-element mappings in earlier iterations. The structural similarity metric is defined as follows:

Definition 4 (Structural Similarity)
Given a feature *f*, a program element *p*, and an edge type *ft* of feature model, let $F = \{f' | \exists p' \in neighbour(p, map(ft)) \ \& \ (f', p') \in M_i\}$ be the set of relevant features of *p*'s neighbouring program elements that are related to *p* through edges of type *map*(*ft*), ICFL measures the structural similarity Sim_S between *f* and *p*, given the edge type *ft*, as follows:

$$Sim_S(f,p,ft) = |F \cap neighbour\ (f,ft)| / |F \cup neighbour\ (f,ft)|$$

The structural similarity Sim_S computes Jaccard coefficient (Tan et al., 2005), indicating how similar two sets, *F* and *neighbour*(*f*, *ft*), are. It essentially measures the intersection of these two sets of features. This intersection set effectively incorporates knowledge of any "known landmarks" (i.e., already established feature-element mappings) for determining the relevance of *f* and *p*.

Let {*et*} be the set of all the edge types defined in the feature model. The structural similarity between *f* and *p* is defined by the following equation:

$$Sim_S(f,p) = \sum_{ft \in \{et\}} Sim_S(f,p,ft) / |\{et\}|$$

3.3.3 Overall similarity metric

Our ICFL algorithm computes the overall similarity metrics between features and program elements using both lexical similarity and structural similarity. Given a feature f and a program element p, ICFL computes their overall similarity metric *Sim* as follows:

$$Sim(f,p) = (1 - w * Conf(f))Sim_L(f,p) + w * Conf(f) * Sim_S(f,p)$$

where w is a weight that defines the extent to which the overall similarity metric depends on the lexical similarity and on the structural similarity, *Conf(f)* is the confidence degree of structural similarity indicating how many of the feature f's neighbouring features can be used as "known landmarks" in the computation of structural similarity. *Conf(f)* is calculated as follow:

$$Conf(f) = |\{f'' \in \cup_{ft\in\{et\}} neighbour(f,ft) \mid \exists p'' \cdot (f'',p'') \in M_i\}| / |\cup_{ft\in\{et\}} neighbour(f,ft)|$$

Conf*(f)* provides another dynamic adjustment to reflect the usability of structural similarity. In the first iteration, *Conf(f)* is equal with 0 and the overall similarity is equal to the lexical similarity between a feature and a program element. Then in the subsequent iterations *Conf(f)* may increase as more and more feature-element mappings have been established.

3.3.4 An example of similarity computation

Take the feature model and the program model in Figure 2 and Figure 3 as an example. Let the already established feature-element mappings be (*BankOpAuth*, *hasBankAuth*), (*IPAddrAuth*, *checkIpAddress*), and (*OperationHistory*, *OperationLog*). Let (*SignIn*, *Operator.login*) be the candidate feature-element pair. Table I summarizes the computation of the structural similarity between the feature *f=SignIn* and the method *p=Operator.login*. Note that the structural similarity for the edge type *read* is 0 and *Conf(SignIn)* is 3/4=0.75 because the mapping (*OperatorList*, *CurOperators*) is not yet established at this moment. Let the weight w be 0.8; let the lexical similarity Sim_L(*SignIn*, *Operator.login*) be 0.52. Then the overall similarity metric *Sim*(*SignIn*, *Operator.login*) is (1-0.8*0.75)*0.52+0.8*0.75*(1+0+1)/3=0.608.

Clearly, the structural similarity between the feature *SignIn* and the method *Operator.login* significantly increases the overall similarity between them than considering only their lexical similarity. This can help to establish the mappings between *SignIn* and *Operator.login*, even if their descriptions are not that similar lexically. Furthermore, the newly established mapping (*SignIn*, *Operator.login*) can in turn increase the structural similarity of the candidate pair (*OperatorList*, *CurOperators*) from 0 to 1, which may further push this pair above the similarity threshold.

TABLE I. STRUCTURAL SIMILARITIES OF (SIGNIN, OPERATOR.LOGIN)

Edge Type	*neighbour(f, ft)*	*neighbour(p, map(ft))*	*F*	Sim_S
decompose/call	BankOpAuth, IPAddrAuth	hasBankAuth, checkIpAddress	BankOpAuth, IPAddrAuth	1
read/data_read	OperatorList	CurOperators	-	0
update/data_write	OperationHistory	OperationLog	OperationHistory	1

3.4 Algorithm

Our ICFL approach recovers traceability links between features and program elements by computing many-to-many feature-element mappings between a feature model and a program model. As a system may consist of thousands of features and program elements, the number of possible many-to-many feature-element mapping solutions can be very large. To avoid exhaustive enumeration of possible solutions, ICFL uses a greedy best-first search algorithm to incrementally construct a "good enough" solution for feature-element mappings.

Our iterative feature location algorithm is described in pseudo code in Algorithm 1. The algorithm takes as input a feature model M_F and a program model M_P. It produces as output a set Map_{F_P} of many-to-many feature-element mappings. Each element of Map_{F_P} is a pair <*f*, *p*> that represents a feature f and its relevant program element p. The set Map_{F_P} is initially an empty set (line 2) and it contains all the feature-element mappings established as the algorithm proceeds finally. The set $Cand_{F\_P}$ contains all the candidate feature-element pairs, i.e., not-yet-mapped feature-element pairs. Initially, it contains all the possible feature-element mappings (line 3).

The algorithm takes four additional parameters, T_{max}, T_{min}, *pace*, and w, which define the upper bound of the similarity threshold, the lower bound of the similarity threshold, the concession pace, and the weight for computing overall similarity metric (see Section 3.3.3). The iterative process starts with *threshold* being set at T_{max} (line 4). In an iteration of feature location, ICFL examines each candidate feature-element pair (*f*, *p*)

in $Cand_{F\_P}$ (line 7), if the overall similarity metric between the feature *f* and the program element *p* is above the current *threshold* (line 8), then ICFL adds this pair of feature-element to the mapping set Map_{F_P} (line 9) and removes it from the candidate set $Cand_{F\_P}$ (line 10), i.e., the feature *f* and the program element *p* has been considered as a pair of established feature-element mapping. If no more new feature-element mappings have been identified in a given iteration (line 14), the algorithm reduces the *threshold* by a *pace* (line 15) and starts a new iteration to identify more feature-element mappings. This process continues until the *threshold* is below the lower bound of similarity threshold T_{min} (line 5). Finally, ICFL algorithm returns the set Map_{F_P} of feature-element mappings (line 18).

We illustrate the iterative feature location process of Algorithm 1 with an example (see Table II) based on the feature model and the program model given in Figure 2 and Figure 3. The algorithm starts with the following settings: T_{max}=0.8, T_{min}=0.4, *pace*=0.2 and *w*=0.8. It ends after four iterations. Table II shows the overall similarity (*Sim*) of the candidate feature-element pairs, the similarity *threshold* used in each iteration, and the set Map_{F_P} after each iteration. As discussed in Section 3.3.3, the similarity of each candidate feature-element pair in the first round is actually its lexical similarity, since no feature-element mappings have been established. After the first iteration, three candidate pairs with overall similarity higher than the threshold (0.8), i.e., (*BankOpAuth*, *Operator.hasBankAuth*), (*IPAddrAuth*, *Operator.checkIpAddress*), (*OperationHistory*, *OperationLog*), have been identified as feature-element mappings.

In the second iteration, as discussed in Section 3.3.4, due to three established feature-element mappings, the similarity of the candidate feature-element pair (*SingIn*, *Operator.login*) increases from 0.52 to 0.608. However, its similarity is still lower than the threshold (0.8). Furthermore, the similarity of (*OperatorList*, *CurOperators*) remains unchanged (0.41). As a result, no new feature-element mappings are established in the second iteration. ICFL then decreases the similarity threshold by 0.2 (from 0.8 to 0.6) and starts the third iteration. Due to the decrease of the similarity threshold, the similarity of (*SingIn*, *Operator.login*) is now above the similarity threshold (0.6) of the third iteration, and thus has been identified as a new feature-element mapping and added into Map_{F_P}. In the fourth iteration, this newly established mapping (*SingIn*, *Operator.login*) increases the structural similarity of the not-yet mapped feature-element pair (*OperatorList*, *CurOperators*), which pushes the overall similarity of (*OperatorList*, *CurOperators*) to 0.882. Thus, (*OperatorList*, *CurOperators*) has been identified as a new mapping and added into Map_{F_P} in the forth iteration.

Algorithm 1. The iterative feature location algorithm
procedure
begin
1. Map_{F_P} ICFL(M_F, M_P, T_{max}, T_{min}, *pace*, *w*)
2. Map_{F_P} = ! ;
3. $Cand_{F\_P}$ = {{*f*, *p*}| *f*! M_F and *p*! M_P};
4. *threshold* = T_{max};
5. while (*threshold* ≥ T_{min} && $Cand_{F\_P}$ ≠ !) {
6. *moreMappingIdentified* = false;
7. for each (*f*, *p*) ! $Cand_{F\_P}$ {
8. if(*Sim*(*f*, *p*) > *threshold*) {
9. Map_{F_P} = Map_{F_P} ! {(*f*, *p*)};
10. $Cand_{F\_P}$ = $Cand_{F\_P}$ \ {(*f*, *p*)};
11. *moreMappingIdentified* = true;
12. }
13. }
14. if(!*moreMappingIdentified*) {
15. *threshold* = *threshold* – *pace*;
16. }
17. }
18. return Map_{F_P};
end

It is important to note that if we were considering only lexical similarity of features and program elements and analyzing features and program elements independently in a single iteration, we would not be

able to identify the feature-element mappings of (*SingIn*, *Operator.login*) and (*OperatorList*, *CurOperators*). In contrast, ICFL takes into account the structural context of features and program elements and propagates the relevance of established feature-element mappings in an iterative process. This allows ICFL to identify more feature-element mappings, i.e., achieve better recall. As we will demonstrate in the empirical studies (Section 4), due to the consideration of structural context in an iterative process, the increase in recall does not sacrifice the precision significantly.

TABLE II. EXAMPLE OF THE ITERATIVE FEATURE LOCATION PROCESS

		Iteration 1	**Iteration 2**	**Iteration 3**	**Iteration 4**
Sim	pair1	0.52	0.608	0.608	-
	pair2	0.85	-	-	-
	pair3	0.91	-	-	-
	pair4	0.41	0.41	0.41	0.882
	pair5	0.93	-	-	-
threshold		0.8	0.8	0.6	0.6
Map_{F_P}		pair2; pair3; pair5	pair2; pair3; pair5	pair1; pair2; pair3; pair5	pair1; pair2; pair3; pair4; pair5

pair1=(*SignIn*, *Operator.login*);
pair2=(*BankOpAuth*, *Operator.hasBankAuth*);
pair3=(*IPAddrAuth*, *Operator.checkIpAddress*);
pair4=(*OperatorList*, *CurOperators*);
pair5=(*OperationHistory*, *OperationLog*)

4. Empirical Studies

To evaluate the effectiveness of ICFL, we conducted empirical studies with two subject systems, DirectBank, a small-scale financial system, and Linux kernel, a large-scale open source operating system kernel. We use DirectBank because it is from our industrial partner. The system experts of DirectBank are available to verify the validity of the results of ICFL. We use Linux kernel because it has well-documented feature models and the ground truth of feature-element mappings can be extracted from the system for evaluation. This allows us to systematically investigate the scalability of our ICFL approach and the quality of ICFL's feature location results.

We use precision and recall metrics to evaluate the effectiveness of the proposed approach in identifying the mappings between features and their relevant program elements. Given a set of features F and a set of program elements P, let Map_{actual} be the set of actual feature-element mappings and Map_{F_P} be the set of feature-element mappings reported by our approach. Precision is the percentage of the correctly reported feature-element mappings $P = (Map_{F\_P} \cap Map_{actual})/Map_{F\_P}$ and recall is the percentage of feature-element mappings reported correctly $R = (Map_{F\_P} \cap Map_{actual})/Map_{actual}$. To reflect a weighted average of the precision and recall, we also use F-measure, which can be computed as $(1 + b^2)/(1/P + b^2/R)$. In our studies, following the treatment in (Hayes et al, 2006), we set b to 2, i.e., recall is considered four times as important as precision, since finding missing traceability links is usually more difficult than removing incorrect links.

4.1 Results for DirectBank

DirectBank is a small-scale industrial database-centric information system written in Java. In this study, we evaluate the effectiveness of ICFL and compare it with an IR-based approach to feature location (Zhao et al., 2006) augmented with incremental threshold concession.

4.1.1 Study Setup

The system experts of DirectBank manually construct the feature model of the system. The feature model involves 71 functional features (such as *SignIn*) and business objects (such as *OperatorList*). The system experts provide the natural language description of these features and business objects. In addition to the decompositions of functional features, features may manipulate (read or update) business objects in the feature model. That is, this feature model has three types of feature dependencies, i.e., decomposition, read and update.

In this study, we would like to identify classes, methods and database tables that implement the features and business objects of DirectBank. We use static program analysis to obtain the program model. We first use Eclipse JDT/DOM APIs to get the Abstract Syntax Tree (AST) model of the program, and then extract program elements (classes and methods) and their call/read/update dependencies from the AST model. The read/update dependencies between the methods and the db_tables are extracted by statically analyzing the SQL queries in JDBC statements.

The system experts define the mapping schema between the edge types of feature model and program model as follows: decompositions correspond to method calls, i.e., feature decompositions are implemented as method calls in DirectBank; read (or update) dependencies correspond to read (or update) between methods and classes/db_tables. Furthermore, the system experts also manually provide the ground truth mappings between the features and business objects and the relevant program elements (classes, methods, and database tables).

We process the description attributes of the features and the program elements in a similar way to other information-retrieval techniques (Zhao et al., 2006), and then encode them in TF/IDF vectors associated with the corresponding features and program elements. As discussed in Section 3.4, ICFL employs an iterative mapping process with threshold concession, so threshold setting is not so sensitive for ICFL but a high initial similarity threshold is required to ensure that the identified feature-element mappings in the initial iterations are most likely to be correct mappings. Therefore, in this evaluation, we set T_{max}=0.8, T_{min}=0.5 and *pace*=0.1. To reflect the importance of structural similarity, we set the weight *w* for computing overall similarity metric at 0.8.

4.1.2 Results

The iterative feature location algorithm executes seven iterations. Figure 4 summarizes the experiment results. The horizontal axis represents the threshold adopted in each iteration, while the vertical axis represents the precision and recall in identifying feature-element mappings in each iteration. Note that in Figure 4 we do not show the iterations during which no new feature-element mappings have been identified. This result suggests that our approach can significantly improve the recall of feature location (from 56.31% to 91.26%) with a slight decrease in precision (from 59.18% to 58.39).

We also apply an IR-based approach to feature location (Zhao et al., 2006) to the DirectBank system. To compare with the iterative process of ICFL, we apply the IR-based approach at the four different thresholds that ICFL used in its iterative mapping process. Note that the application of IR-based approach at different thresholds is independent of each other, because the IR-based approach cannot iteratively use feature-element mappings established in earlier rounds. Figure 5 presents the result. Compared with the proposed ICFL approach, this IR-based approach significantly sacrifices the precision (from 59.18% to 22.93%) in order to improve the recall (from 56.31% to 80.58%) at the lower similarity threshold. We attribute the better performance of our approach to its iterative context-aware nature, which makes it less sensitive to the choice of the similarity threshold and the quality of the lexical descriptions of features and program elements.

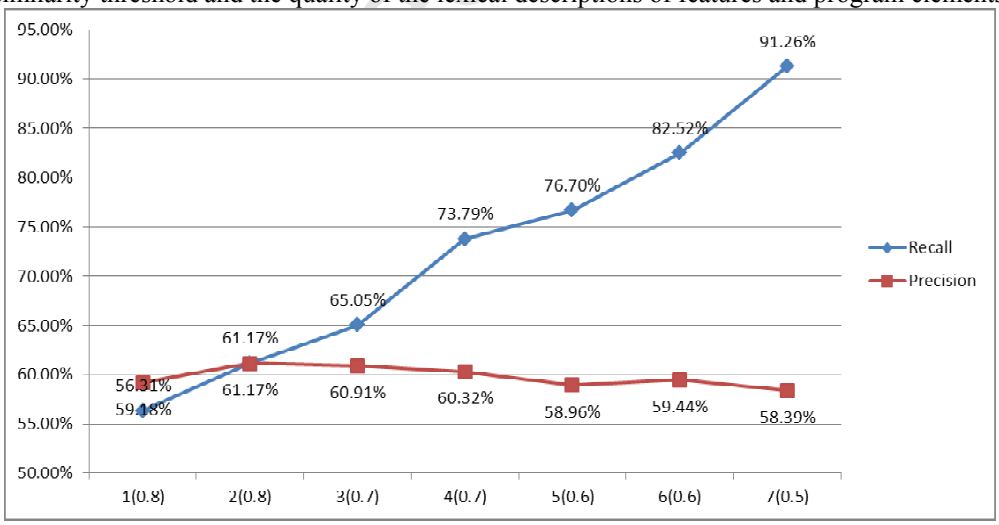

Figure 4. Precision/recall of ICFL with DirectBank

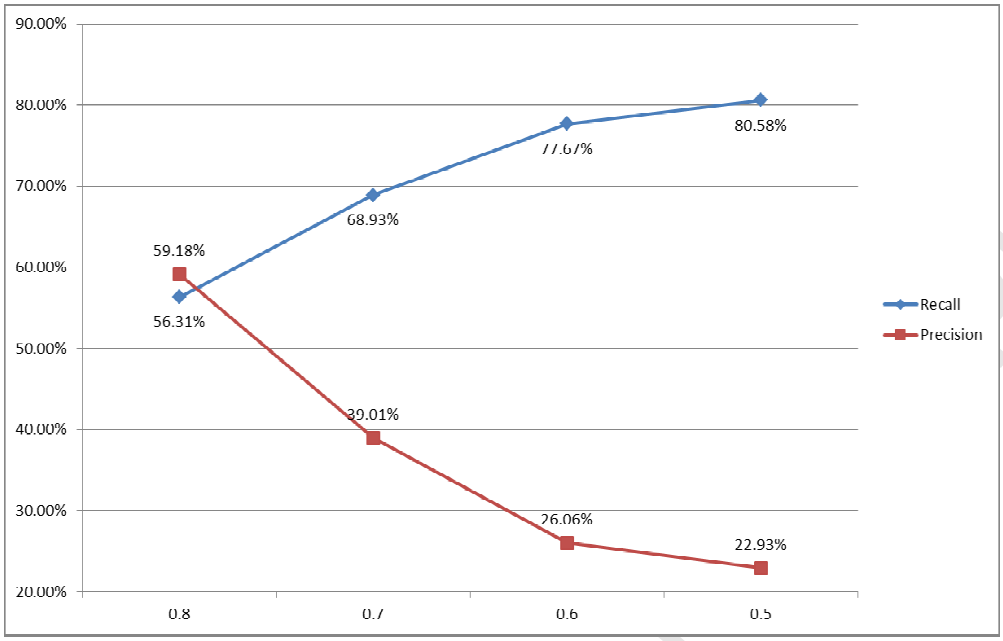

Figure 5. Precision/recall of an IR-based FL with DirectBank

4.2 Results for Linux kernel

Linux kernel is a large-scale open source operating system kernel written in C. In this study, we evaluate the effectiveness of ICFL and compare it with an IR-based approach to feature location (Zhao et al., 2006) augmented with incremental threshold concession.

4.2.1 Subject System

The Linux kernel system was originally developed for 32-bit x86-based PCs, but has since been ported to numerous architectures, such as Alpha, SPARC, PowerPC, and also mainframes and handheld devices. In fact, the Linux kernel has become a software product line (Berger et al., 2010; She et al., 2010), which consists of thousands of features that can be configured in order to generate specific kernel products for vast combinations of architectures, subsystems, device drivers, etc.

The Linux kernel manages these features and their variability using a feature modeling language (Kconfig) and its accompanied configuration tool (Berger et al., 2010; She et al., 2010). The configuration tool allows users to initialize and modify a configuration. It supports a few initialization options, for example, using the default configuration for a given architecture (defconfig); randomly selecting features to be included in the kernel product (randconfig); or selecting as many features as possible (allyesconfig). The user then uses GUI-based configuration tool to modify the configuration to reach the desired state.

This configuration process results in a valid selection of features expressed as a set of configuration symbols, such as CONFIG_PCI, which represents the feature of PCI supporting in the Linux system. The Linux kernel uses C preprocessor (CPP) language to handle feature implementations at source code level. The configuration symbols are used in Makefiles and source code to control which directories, files, and conditional blocks are selected for compilation. In fact, the usage of these configuration symbols in Makefiles and source code provides the ground truth mappings (as defined by Linux kernel developers) of features and their relevant program elements (e.g. source files, functions).

4.2.2 Experiment Setup

In this study, we use the version 2.6.36.2 of Linux kernel that was released on December 9th, 2010 as the subject system to evaluate our ICFL approach.

The Kconfig files defining features and feature dependencies can be considered as a Linux-kernel specific feature model (Berger et al., 2010; She et al., 2011). Therefore, in this study we extract features and feature dependencies from Kconfig files to construct the feature model of Linux kernel as feature-level input for ICFL. The resulting feature model of Linux kernel 2.6.36.2 consists of 6052 features. Note that not all these 6052 features can be included in a specific kernel product on a given system architecture. In this study, we generate a kernel configuration for the x86_32 system architecture using allyesconfig profile (i.e.,

selecting as many feature as possible). Furthermore, in this study we consider only the core kernel features that are implemented in the following core kernel subsystems:

- *arch/x86* contains the specific kernel code for x86-based architecture (Our experiment was conducted on a 32-bit x86-based PC. We did not consider other hardware architectures/CPUs in this study).
- *kernel* contains the main architecture-independent kernel code.
- *include* contains most of the system's include (.h) files.
- *fs* contains support for various kinds of file systems.
- *ipc* contains the code for inter-process communications.
- *mm* contains the architecture-independent memory management code.
- *lib* contains the architecture-independent library code.
- *net* contains the networking support for sockets and TCP/IP (Code for particular networking cards is contained in drivers/net).
- *block* contains the block device code.
- *sound* contains two kinds of sound supports, i.e., alsa and oss.
- *crypto* contains support for hardware crypto devices.
- *init* contains the system initialization code.
- *security* contains Linux security module code.

Linux kernel 2.6.36.2 contains in total approximately 12 million lines of code, 2,084 directories, 28,024 files and 155,266 C functions. The above core subsystems that we consider in the study involve about 18.0% (1088/6052) of features of the Linux kernel and about 10.2% (2871/28024) of source (.c) files of the full source tree. We decide to omit device drivers (e.g. sound, PCI, networking card, SCSI) in this study. These device drivers are usually small and self-contained. It is relatively easy to determine the implementation of these device drivers.

We develop a set of program analysis tools to construct the feature model from Kconfig files and the program model from source code. The Linux kernel is an operating system kernel. It does not deal with business objects and database. Its feature model consists of only *functional features* and *decomposition* relationships between features. In this study, we would like to identify source files (.c) that implement the functional features of Linux Kernel. Therefore, the reverse-engineered program model consists of source files and their usage dependencies. Note that the usage dependencies between source files are aggregated from the function calls between functions defined in the corresponding source files.

In this study, the mapping schema between edge types of feature model and program model is relative simple, i.e., feature decompositions correspond to file usage dependencies. Furthermore, we develop a tool to analyze the usage of feature configuration symbol (i.e., CONFIG_xxx) in makefiles and C preprocessors. This tool extract the ground truth mappings (as defined by Linux kernel developers) of features and their relevant program elements from Linux kernel. For Linux kernel version 2.6.36.2, the extracted ground truth consists of in total 3990 feature-element mappings. In this experiment, we also set the threshold T_{max} as 0.8, T_{min} as 0.5, *pace* as 0.1 and *w* as 0.8.

4.2.3 Runtime Performance

The empirical study with Linux kernel is conducted on a computer with Windows Server 2008, Intel Xeon 2.13GHz, 32GB RAM. As Linux kernel is a large-scale system with thousands of features and source files, the iterative mapping process of ICFL algorithm is quite time-consuming. Therefore, to speed up the iterative mapping process, we introduce one more filtering threshold F_{min} that is set to 0.2 in this study. If the overall similarity metric of a candidate feature-element mapping is less than F_{min}, the candidate mapping will be removed from the candidate set $Cand_{F\_P}$.

After taking this filtering strategy, it takes about 3 hours for our ICFL algorithm to complete the 18 iterations of the mapping process (see Figure 6) for locating the implementation files of all 1088 features. The filtering strategy may affect the precision and recall of the feature location results of ICFL algorithm because it ignores some candidate mappings. However, it can significantly improve the runtime performance of our ICFL algorithm, while its impact on the quality of the feature location results is minor as we discuss below.

4.2.4 Results

Again, we use precision and recall metrics to evaluate the quality of the feature-element mappings reported by our ICFL algorithm in Linux kernel study. Figure 6 summarizes the experiment results. The X-

axis represents the iterations and the threshold adopted in each iteration, and the Y-axis represents the precision and recall in each iteration.

From Figure 6, it can be seen that depending on structural contexts of features and source files ICFL significantly improves the recall (from 7.44% to 66.03%) of feature-element mappings with a decrease of precision (from 46.28% to 29.52%). Overall, ICFL makes a significant improvement on F-measure metrics (from 0.089 to 0.529). Considering the significant improvement on recall and F-measure we believe the decrease of precision is acceptable. It is interesting to note that in the initial four iterations the recall increases only slightly from 7.44% to 9.34%, with about 5% decrease in precision. However, as more and more feature-element mappings have been established and contributed to the similarity computation of not-yet mapped feature-element mappings, the recall increases sharply (about 56%) in the following iterations with only minor (about 11%) decrease in precision.

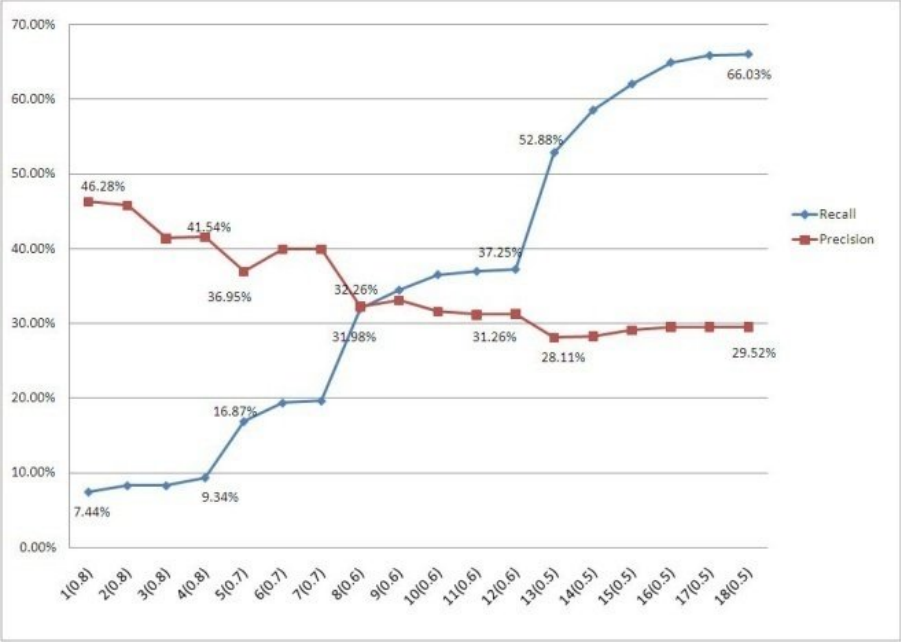

Figure 6. Precision/recall of ICFL with Linux kernel

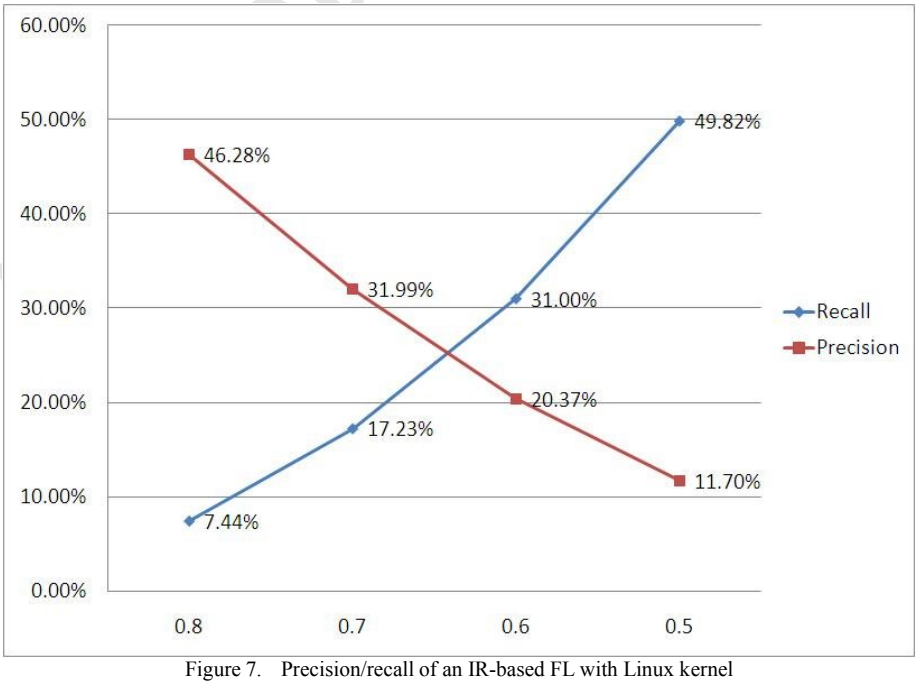

Figure 7. Precision/recall of an IR-based FL with Linux kernel

We also apply an IR-based approach to feature location (Zhao et al., 2006) to the Linux kernel data. Similar to the study with DirectBank, we apply the IR-based approach at the four different thresholds that ICFL adopted to facilitate the comparison with the iterative process of ICFL. Figure 7 presents the experiment results. Clearly, without considering structural contexts of features and program elements in an iterative process, the decrease of similarity threshold significantly sacrifices precision (from 46.28% to 11.70%) for recall (from 7.44% to 49.82%). Even with this significant sacrifice of precision, the best recall (49.82% at the similarity threshold 0.5) that this IR-based approach achieves is still much worse than the recall (66.03%) that our ICFL approach achieves at the same similarity threshold (0.5). Thus, although at lower threshold the IR-based approach also achieves better F-measure metrics (improved from 0.089 to 0.302), it is clear that the improvement is much lower than that of ICFL approach (0.302 versus 0.529). Furthermore, ICFL manages to keep precision in an acceptable range. The worst precision (29.52% at the similarity threshold 0.5) of ICFL in this Linux kernel study is much higher than the worst precision (11.70%) of the IR-based approach. According to Figure 7, if this IR-based approach wants to achieve the precision (29.52%), its recall would be only about 20%, which is significantly worse than the recall (i.e., 60.52%) of ICFL at the same precision value.

Let us take a closer look at the feature location results of our ICFL algorithm and the IR-based approach for a specific Linux kernel feature "The wireless configuration of Linux system". For the sake of simplicity, we refer to this Linux kernel feature as "*CONFIG_80211*" (i.e., the configuration symbol of this feature used in Kconfig file) in the following discussion. Table III summarizes the source files (.c) that are relevant to this specific Linux kernel feature, i.e., the ground truth mappings between the relevant source files and the feature *CONFIG_80211*. As shown in the Table, the feature *CONFIG_80211* has been implemented in 15 different source (.c) files.

Table III also presents the lexical similarity metrics, the final structural similarity metrics and the final overall similarity metrics between the feature *CONFIG_80211* and each of its implementing source file. The last two columns show the files that can be identified as relevant program elements by ICFL and the IR-based approach with the similarity threshold of 0.5 respectively. Because the description of the feature *CONFIG_80211* is very short, the lexical similarity metrics (the column Sim_L) between its descriptions and the relevant source files are relative low. However, by taking into account the similarity of the structural contexts (the column Sim_S) of the feature *CONFIGU_80211* and the relevant source files, the overall similarity metrics (the column *Sim*) have been significantly improved. If the similarity threshold is set to 0.5, our ICFL algorithm can successfully identify 66.7% (10/15) of feature-element mappings, while the IR-based approach can only find 20% (3/15).

TABLE III. RESULTS FOR THE FEATURE "THE WIRELESS CONFIGURATION OF LINUX SYSTEM"

file_id	*file_name*	Sim_L	Sim_S	*Sim*	*ICFL*	*IR-based Approach*
7143	net/wireless/ibss.c	0.51	0.80	0.74	√	√
7194	net/wireless/radiotap.c	0.37	0.00	0.07		
7202	net/wireless/scan.c	0.36	1.00	0.87	√	
7207	net/wireless/sme.c	0.45	0.80	0.73	√	
7214	net/wireless/core.c	0.38	1.00	0.88	√	
7229	net/wireless/util.c	0.46	0.80	0.73	√	
7249	net/wireless/chan.c	0.48	0.40	0.42		
7250	net/wireless/wext-sme.c	0.49	0.80	0.74	√	
7251	net/wireless/reg.c	0.39	1.00	0.88	√	
7261	net/wireless/nl80211.c	0.37	0.00	0.07		
7267	net/wireless/sysfs.c	0.33	1.00	0.87	√	
7275	net/wireless/ethtool.c	0.50	0.00	0.10		√
7276	net/wireless/mlme.c	0.33	1.00	0.87	√	
7278	net/wireless/wext-compat.c	0.52	0.60	0.58	√	√
7281	net/wireless/debugfs.c	0.33	0.20	0.23		

4.3 Summary

From the results with DirectBank (Figure 4 and Figure 5) and Linux kernel (Figure 6 and Figure 7), it can be seen that precision tends to decrease as recall increases except a few exceptions in the experiments using ICFL, for example the second and sixth iterations in Figure 4 and the fourth and sixth iterations in Figure 6. These exceptions were caused by the fact that the similarity threshold did not decrease in these iterations while more correct mapping pairs were accepted due to the increase of structural similarities of matching candidates.

The iterative process of the ICFL approach suggests that there are two key factors that cause the increase of recall, i.e., more candidate pairs being accepted as mappings. One factor is the decreased threshold, and the other is the availability of more and more mapped structural relations. Increase of recall resulted from the former factor is completely earned by sacrificing precision. In contrast, increase of recall resulted from the latter factor is achieved by utilizing structural similarity provided by already established mappings, thus has minor influence on precision. Therefore, in the two IR-based experiments (Figure 5 and Figure 7), in which no structural similarity is taken into account, the precision decreases rapidly as the recall increases. Furthermore, we can see that compared with the relative bigger decrease of precision in Linux kernel study (from 46.28% to 29.52% in Figure 6), there is only a very slight decrease of precision in DirectBank study (from 59.18% to 58.39 in Figure 4). Our further analysis suggests that this difference is due to the higher-quality of initial mappings established in the first iteration and also due to the fact that features and program elements of the DirectBank system has richer structural context, i.e., more node and edge types in both the feature model and the program model. The high-quality initial mappings and the richer structural context help to reduce the impacts of lower similarity threshold on the mapping process and thus prevent erroneous feature-element mappings from being established.

5. Discussion

Our empirical study shows that the structural similarity between features and program elements can effectively improve the quality of feature location results in an iterative graph matching process. However, the effectiveness of our ICFL approach may be affected by several factors, including the quality of the initial mapping results in the first iteration, and the concession pace of similarity threshold in the iterative mapping process, and the tradeoffs between the quality (precision and recall) of the feature location results and the human effort required when applying ICFL approach in practice.

5.1 The impact of initial mapping results in the first iteration

Our ICFL approach exploits the structural contexts of features and program elements for the purpose of feature location. For example, if a feature is decomposed into a few sub-features, then the relevant program elements implementing these features may manifest certain kinds of usage dependencies (such as method calls). The similarity of such structural contexts of features and program elements are inherent in the system. In contrast, the lexical descriptions of feature and program elements are more ad-hoc and uncertain. It requires much effort to "normalize" in order to make these lexical descriptions useful (Butler et al., 2011).

Nevertheless, the computation of structural similarity requires certain established mappings of features and program elements in the structural contexts of candidate features and program elements. As a result, the initial mapping results of the first iteration of ICFL may affect the subsequent iterations of ICFL. Take the experiment results of DirectBank and Linux kernel shown in Figure 4 and Figure 6 as an example. Compared with the overall precision and recall that ICFL achieves in the DirectBank study (see Figure 4), it can be seen that the overall precision and recall that ICFL achieves in the Linux kernel study (see Figure 6) are worse than those of DirectBank. Linux kernel is a much more complex system than DirectBank. It supports thousands of features implemented in tens of thousands of source files. This significantly increased system complexity affects the quality of feature location results of ICFL.

Furthermore, it is interesting to note that the recall (7.44%) and precision (46.28%) in the first iteration of ICFL on Linux kernel data is much lower than the recall (56.31%) and precision (59.18%) in the first iteration of ICFL on DirectBank data. We believe that the much lower precision and recall in the first iteration also affect the overall precision and recall of ICFL in the Linux kernel study. We find that this worse first iteration on Linux kernel data is mainly due to the short feature descriptions extracted from Kconfig files (about 48.43% of the features contain less than 30 words). Consequently, it makes it difficult to determine the relevance between features and program files in the first iteration during which only lexical similarities are available as no feature-element mappings have been established yet. In contrast, the feature descriptions in the DirectBank study are provided by the system expert. These descriptions are much more

complete and precise, which results in a better precision and recall in the first iteration. This better first-iteration precision and recall in turn provides a good basis for the subsequent iterative mapping process.

This indicates that the quality of lexical descriptions of features and program elements may still affect the quality of feature location results of our ICFL approach, especially in the first few iterations. However, as we discuss in the comparison of our ICFL results and those of IR-based approach (Figure 4 versus Figure 5, and Figure 6 versus Figure 7), the final results of our ICFL is much less sensitive to the quality of lexical descriptions of features and program elements than IR-based approach, because our ICFL takes into account structural contexts of features and program elements in an iterative mapping process.

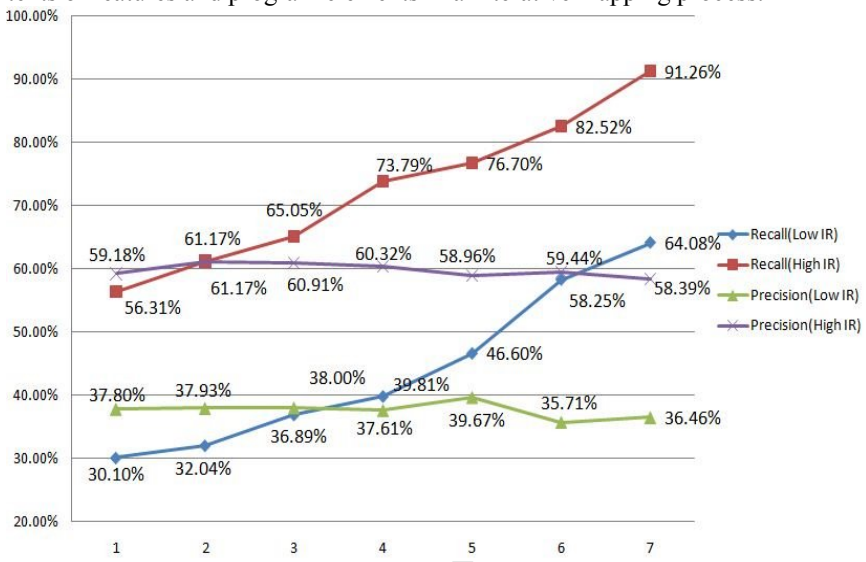

Figure 8. Precision/recall of ICFL with different initial mapping quality (DirectBank)

In order to evaluate the impact of the initial mapping results on the subsequent iterations of ICFL, we conduct another experiment on DirectBank system, in which we intentionally use less complete or precise feature descriptions. As a result, the lexical similarities between DirectBank features and program elements are significantly decreased, which in turn greatly affect the quality of the initial mapping results in the first iteration. Figure 8 presents the results of the second experiment on DirectBank. The lower two lines present the precisions and recalls of this second experiment on DirectBank. Compared with the precisions and recalls of the first experiment on DirectBank (the higher two lines), it can be seen that the quality of the initial mapping results greatly affects the precision and recall of the final feature location results. Given the worse initial mapping results, our ICFL algorithm can still improve the quality of feature location results in the subsequent iterations. However, the quality of its final results can be significantly worse than the quality of the final feature location results based on better initial mapping results.

Therefore, to produce high-quality mapping results in the first iteration, one should try to provide more complete and precise description for each feature. Furthermore, a high initial threshold (e.g., 0.8 or higher) is often useful to ensure the high-quality of initially accepted mapping pairs.

5.2 The concession pace of similarity threshold

In the feature location process of ICFL algorithm (see Algorithm 1), a high initial similarity threshold (i.e., T_{max}) is necessary to ensure that the identified feature-element mappings in the initial iterations are most likely to be correct mappings. In additional to T_{max}, the pace of threshold concession is another important input parameter. Intuitively, a small concession pace value decreases the similarity threshold in small steps, thus can potentially affect the precision and recall of identified feature-element mappings by enforcing more mapping iterations. In these iterations, more feature-program-elements pairs can be activated and used for similarity computation. In other words, with a smaller concession pace, structural similarity plays more important role in the computation of the overall similarity metric (see Section 3.3.3) and also the selection of feature-program mappings.

Figure 9 presents our experiment results of the precisions and recalls of the final feature location results of ICFL on DirectBank system using different concession paces of similarity threshold. The x-axis shows different concession paces (ranging from 0.01 to 0.2), while the y-axis shows the precision and recall values

of the final feature locations results with different concession paces. For example, using the concession pace 0.01, the precision and recall of the final feature location results of ICFL is 61.81% and 86.41%, respectively. Overall, our experiment results show that as the concession pace increases (or decreases), the precision may decrease (or increase) slightly with a slight increase (or decrease) in recall.

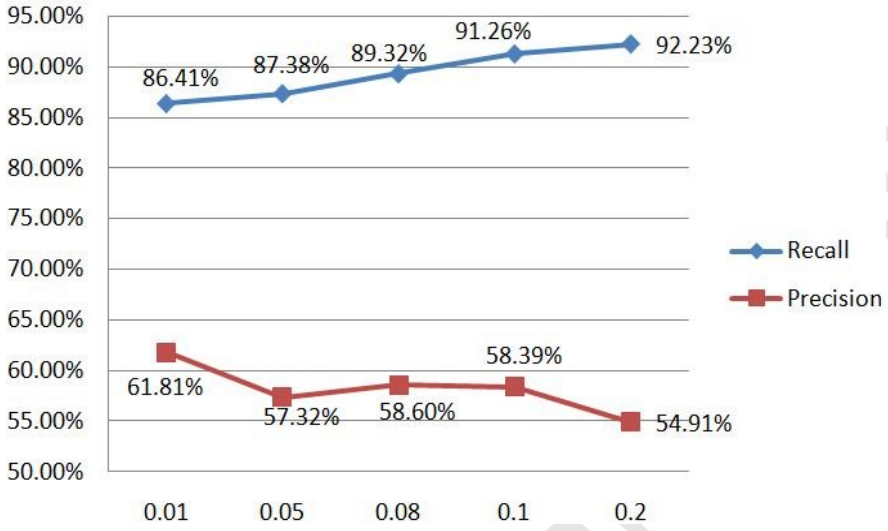

Figure 9. Evaluation of the final precision and recall with Different concession pace (DirectBank)

However, smaller concession pace of similarity threshold may result in worse runtime performance due to more mapping iterations of ICFL. For example, in DirectBank study ICFL ends after five iterations using concession pace 0.2, while it executes 27 iterations using concession pace 0.05. Consequently, the execution time of the ICFL algorithm increases from half an hour to twenty hours. According to our experience, for small-scale or medium-scale systems like DirectBank a concession pace between 0.05 and 0.1 may be appropriate, while for large-scale systems like Linux Kernel a concession pace between 0.1 and 0.15 may be appropriate considering the performance issue.

5.3 Crosscutting features

Many features in software systems are crosscutting. A crosscutting feature may mean that the feature itself is crosscutting, or its implementation is crosscutting.

A business expert can identify a feature to be crosscutting if it interacts with multiple other features. As shown in Figure 10, this kind of crosscutting features (e.g., *Log*) and their interactions with other features can be represented as feature relations in a feature model. In our ICFL approach, such relations between features do not necessarily form a strict tree model. For example, a crosscutting feature *Log*, which interacts with several other features such as *OrderPayment* and *OrderChecking*, can be represented by *use* dependencies among these features. These *use* dependencies, if mapped to corresponding program relations (e.g., method *calls*), can also provide the structural contexts required for locating the *Log* feature in the implementation

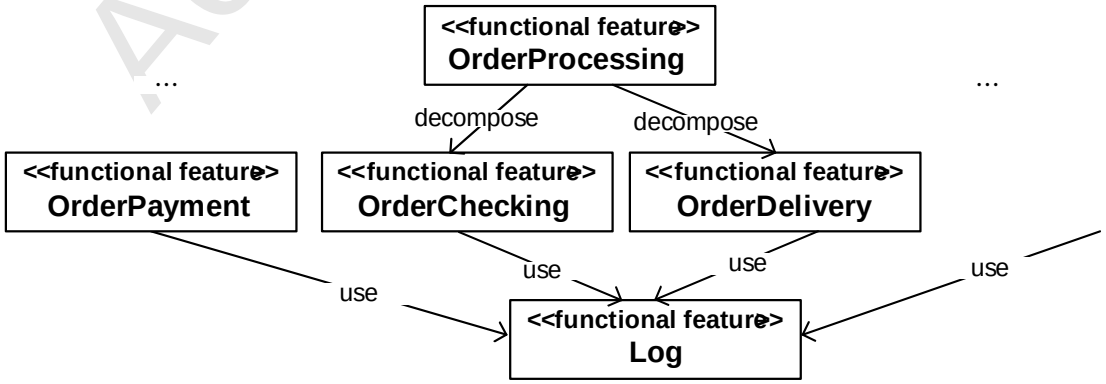

Figure 10. An example of crosscutting feature

From the implementation perspective, code units that are relevant to a crosscutting feature may be scattered in several modules. Take the *Log* feature again as an example. It is often the case that in addition to a logging method there are also other relevant program units scattered in modules corresponding to other features such as *OrderPayment* and *OrderChecking*. Feature location is the activity of identifying an initial location in the source code that implements a given feature, while the full extent of all the code relevant to a feature is handled by impact analysis (Dit et al., 2011). In other words, feature location in a narrow sense means to identify specific program elements that contribute to a given feature's implementation and not to any other features' implementation (Zhao et al., 2006). Therefore, we believe that, if we locate only specific program elements for a given feature, the structure of source code should resemble the structure of features in general. Based on the specific program elements located by ICFL, we can extend the results with more relevant program elements, for example, by exploring program call graph as what has been done in SNIAFL (Zhao et al., 2006).

Having that said, we believe that the modularity of a software system's implementation has a significant influence on the effectiveness of the ICFL approach. For a software system that has a good modularity design and follows the principle of separation of concern, its program structure will align well with the structure of features, thus allows the successful application of our ICFL approach.

5.4 Practical use of the ICFL approach

When using our ICFL approach in practice, developers are required to provide a feature model as an input. The feature model can be manually constructed by domain experts or (semi-)automatically extracted from product documents (She et al., 2011; Dumitru et al., 2011; Yang et al., 2009). Therefore, certain amount of human effort may be necessary for constructing feature models, especially when the system does not have a feature model or other similar form of feature documents. For example, in the study of Linux kernel system we can automatically extract the feature model of Linux kernel from the accompanied Kconfig files of the Linux kernel, while in the study of DirectBank we have to resort to the system experts of DirectBank who manually constructed the required feature model for our empirical study.

Furthermore, the feature models can be coarse-grained (Kang et al., 1990), for example including only coarse-grained features and decomposition relations. The feature models may also be fine-grained (Zhang et al., 2006; Peng et al., 2006), for example defining not only fine-grained features and decomposition relations but also additional feature dependencies. Fine-grained feature models usually require more human effort to construct. However, fine-grained feature models allow our ICFL algorithm to produce better feature location results, because they provide richer structural contexts of features that can be exploited in the feature location process of ICFL. In our two empirical studies on DirectBank and Linux kernel, we use only limited types of feature dependencies, such as feature decomposition. However, it is important to note that our ICFL approach is not limited to those types of feature (and program-element) dependencies used in our empirical studies. In fact, ICFL can be easily configured to exploit (open-ended) types of feature dependencies and/or program-element dependencies (see Definitions in Section 3.3.2).

It should be noted that the feature model required by the ICFL approach can be reused through the maintenance process and shared among developers of the same system. After the initial construction, the feature model can be updated and optimized based on requirements changes and developers' feedback. Thus the feature model can be maintained and shared as a common knowledge base facilitating developers' maintenance tasks. In addition to reusing feature model, experiences about proper parameter settings of ICFL algorithm can also be shared among developers of the same system.

Given the feature location results, a developer may still need to validate the reported feature-element mappings to identify false positives. As our ICFL approach adopts an iterative process with threshold concession, mapping pairs produced by ICFL may be accepted in different iterations with different threshold. The reported mappings can be ranked by their similarity measurements computed on both lexical and structural similarity. The developer then can check the results according to the ranked list of feature-element mappings. Our experience shows that mappings reported in earlier iterations usually have better quality, i.e., with less false positives. On the other hand, the quality of feature location results can also be improved with developers' feedback when they check the results. ICFL can be used in an iterative manner by requesting developers' feedback after each iteration. In this way, the iterative process can be ended according to the quality feedback from developers instead of the predefined threshold. Moreover, as a large number of false positives can be eliminated according to developers' feedback, the quality of feature location results can be improved with more accurate contextual information.

5.5 Threats to validity

The major threats to the validity of our studies lie in the following three aspects.

The first threat is the quality of the ground truth mappings used to evaluate the quality of feature location results. For the DirectBank study, the system experts from our industrial partner manually identified the ground truth mappings. DirectBank is a small system and its system experts had maintained the system for years. Thus, we believe that the ground truth mappings they provided are reliable. For the Linux kernel study, the ground truth mappings are automatically extracted based on the usage of feature configuration symbols in Makefiles and source code as defined by Linux kernel developers. Overall, we believe the threat to the validity of ground truth mapping is acceptable for our empirical studies.

The second threat lies in the fact that our ICFL approach requires a predefined feature model and corresponding mapping schema. As discussed in Section 5.4, the granularity and quality of the predefined feature model can influence the quality of feature location results. On the other hand, the granularity of program information may also influence the quality of feature location results. For example, we used Java methods (also classes) in the DirectBank study and source files in the Linux kernel study respectively. Our ICFL approach requires a mapping schema between the edge types of feature model and program model (see Definition 3 in Section 3.3.2) so that it can determine the similarity of the structural context of features and program elements. However, because feature model and program model describe the system at different levels of abstraction, it is not always straightforward to determine the correspondence between the edge types between the two types of models. For example, a feature decomposition relation may be implemented by method/function call or message passing. One may have to understand the specific style of system design and implementation in order to define appropriate mapping schema between the edge types in feature model and program model.

The third threat is the "greedy-best-first-search-without-backtracking" strategy used in our ICFL algorithm. This strategy may lead to “suboptimal” feature location results, due to the erroneous feature-element mappings identified in the initial iterations of ICFL process. As discussed in Section 5.2, we may alleviate the negative impacts of this search strategy on the quality of feature location results of ICFL by adopting high initial similarity threshold and/or using small concession pace of similarity threshold. The high initial similarity threshold and small concession pace ensure that the feature-element mappings (especially those identified in initial iterations of ICFL) are most likely correct.

6. Conclusion and Future Work

In this paper, we have proposed an iterative context-aware approach to automatic feature location. Taking as input a feature model and a program model, our approach examines both the lexical description and the structural context of the features and the program elements for determining the feature-element mappings in an iterative feature location process. Our experimental study on a small-scale industrial product and a large-scale open source operating system has confirmed that our approach can significantly increase the recall of feature location task with a good control in decrease of precision. Furthermore, our approach is less sensitive to the choice of the similarity threshold and the quality of the lexical description of the features and the program elements.

Our study suggests that current feature location approaches can be improved by utilizing structural context information of features and program elements with application-specific knowledge about the mapping schema between feature relation types and program relation types. The similarity of such structural contexts of features and program elements are inherent in many systems, making the feature location process more robust and less sensitive to the quality of the input (e.g., lexical description of the features and the program elements).

The rationale behind our approach is to utilize the prior knowledge of developers about a target system to improve the effectiveness of feature location techniques. Currently our ICFL approach only employs feature relations for determining the mappings between features and program elements. In the future, we will investigate the usage of design knowledge such as architecture models in our approach. Furthermore, we will conduct systematic empirical studies on more complex and hybrid systems such as multi-layered web systems consisting of web pages, Java classes and XML configuration files.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grant Nos. 60703092 and 90818009, National High Technology Development 863 Program of China under Grant No.

2012AA011202, National Research Foundation for the Doctoral Program of Higher Education of China under Grant No.20100071110031.

References

Adams, B., Jiang, Z., Hassan, A, 2010. Identifying Crosscutting Concerns Using Historical Code Changes. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 305-314.

Baeza-Yates, R., Ribeiro-Neto, B., 1999. Modern Information Retrieval, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Berger, T., She, S., Lotufo, R., WasoWski, A., Czarnecki, K., 2010. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In: Proceedings of the 25th International Conference on Automated Software Engineering, pp. 73-82.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2011. Improving the Tokenisation of Identifier Names. In: Proceedings of the 25th European Conference on Object-Oriented Programming, pp. 130-154.

De Lucia, A., Oliveto, R., Tortora, G., 2008. IR-Based Traceability Recovery Processes: An Empirical Comparison of "One-Shot" and Incremental Processes. In: Proceedings of the 23rd International Conference on Automated Software Engineering, pp. 39-48.

Dumitru, H., Gibiec, M., Hariri, N., Cleland-Huang, J., Mobasher, B., Castro-Herrera, C., Mirakhorli, M., 2011. On-demand feature recommendations derived from mining public product descriptions. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 181-190.

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2011. Feature location in source code: a taxonomy and survey. Journal of Software Maintenance and Evolution: Research and Practice. doi: 10.1002/smr.567.

Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. IEEE Transactions on Software Engineering 29 (3), pp. 210- 224.

Ermolayev, V., Keberle, N., Matzke, W., Vladimirov, V., 2005. A strategy for automated meaning negotiation in distributed information retrieval. In: Proceedings of the 4th International Semantic Web Conference, pp. 201-215.

Hayes, J.H., Dekhtyar, A., Sundaram, S.K., 2006. Advancing candidate link generation for requirements tracing: The study of methods. IEEE Transactions on Software Engineering 32 (1), pp. 4- 19.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., 1990. Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-021, Software Engineering Institue, Carnegie Mellon University, Pittsburgh, PA.

Kazato, H., Hayashi, S., Okada, S., Miyata, S., Hoshino, T., Saeki, M., 2012. Feature Location for Multi-Layer System Based on Formal Concept Analysis. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering, pp. 429-434.

Linux kernel. http://www.kernel.org/.

Marcus, A., Maletic, J.I., 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering, pp. 125-135.

Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I., 2004. An Information Retrieval Approach to Concept Location in Source Code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214- 223.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C., 2011. Portfolio: Finding Relevant Functions and Their Usages. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 111-120.

Peng, X., Zhao, W., Xue, Y., Wu, Y., 2006. Ontology-based Feature Modeling and Application-Oriented Tailoring. In: Proceedings of the 9th International Conference on Software Reuse, pp. 87-100.

Peng, X., Xing, Z., Tan, X., Yu, Y., Zhao, W., 2011. Iterative context-aware feature location: (NIER track). In: Proceedings of the 33rd International Conference on Software Engineering, pp. 900-903.

Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., Rajlich, V., 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Transactions on Software Engineering 33 (6), pp.420-432.

Poshyvanyk, D., Marcus, A., 2007. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In: Proceedings of the 15th IEEE International Conference on Program Comprehension, pp.37-48.

She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K., 2010. The Variability Model of The Linux Kernel. In: Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 45-51.

She, S., Lotufo, R., Berger T., Wa¸sowski, A., Czarnecki, K., 2011. Reverse Engineering Feature Models. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 461-470.

Tan, P.-N., Steinbach, M., Kumar, V., 2005. Introduction to Data Mining, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Warr, F.W., Robillard, M.P., 2007. Suade: Topology-Based Searches for Software Investigation. In: Proceedings of the 29th International Conference on Software Engineering, pp.780-783.

Yang, Y., Peng, X., Zhao, W., 2009. Domain Feature Model Recovery from Multiple Applications using Data Access Semantics and Formal Concept Analysis. In: Proceedings of the 16th Working Conference on Reverse Engineering, pp.215-224.

Zhang, W., Mei, H., Zhao, H., 2006. Feature-driven requirement dependency analysis and high-level software design. Requirements Engineering 11 (3), pp. 205-220.

Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F., 2006. SNIAFL: Towards A Static Noninteractive Approach to Feature Location. ACM Transactions on Software Engineering and Methodology 15 (2), pp. 195-226.

Highlights

We propose ICFL, an iterative context-aware approach for feature location

ICFL considers structural similarity between features and program elements

We evaluate ICFL using a small industry system and a large open-source system

ICFL is more robust and can increase recall with only minor decrease of precision

Structural similarity can complement lexical similarity for feature location