Iterative Context-Aware Feature Location (NIER Track)

Xin Peng[1], Zhenchang Xing[2]
[1]School of Computer Science, Fudan University, Shanghai 201203, China
{pengxin, 09210240034, wyzhao}@fudan.edu.cn

Xi Tan[1], Yijun Yu[3]
[2]School of Computing, National University of Singapore, 10 Kent Ridge Cresent, Singapore 119260
xingzc@comp.nus.edu.sg

Wenyun Zhao[1]
[3]Department of Computing, The Open University, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom
y.yu@open.ac.uk

ABSTRACT
Locating the 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, and they tend to miss the non-local interactions among features. In this paper, we propose to address the proceeding two issues in feature location using an iterative context-aware approach. The underlying intuition is that the 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 their 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. Our initial evaluation suggests the proposed approach is more robust and can significantly increase the recall of feature location with a slight decrease in precision.

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

General Terms
Algorithms, Design, Experimentation

Keywords
Feature location, Information retrieval, Structural similarity.

1. INTRODUCTION
Locating the 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) [4, 7], dynamic analysis [2], and the hybrid of several analysis techniques [5].

ICSE'11, May 21-28 2011, Waikiki, Honolulu, HI, USA.

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. More importantly, the existing techniques for feature location tend to miss the non-local interactions among features. These interactions can be critical to avoiding unexpected side-effects during code modification.

To address the proceeding issues, we propose an Iterative Context-aware approach to automatic Feature Location (ICFL). ICFL assumes that: (1) the features are not independent of each other, for example, a feature may use, extend, or refine other features; (2) the structure of source code resembles the structure of features (see an example in Figure 1 and Figure 2). Thus, ICFL takes as input a requirement model that captures the features and their interdependencies and a program model that captures the program elements and their interdependencies (Section 2.1). It solves the 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 program element based on their lexical and structural similarity (Section 2.2). It employs an iterative process to propagate the knowledge of the established mappings between a feature and a program element to the neighboring features and program elements (Section 2.3). 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.

We evaluate our ICFL approach using a small-scale financial system (DirectBank) from our industry partner. We compare our approach with an IR-based approach to feature location [7]. Our evaluation suggests the proposed iterative context-aware approach to feature location is more robust and can significantly increase the recall of feature-element mappings with a slight decrease in precision. This evaluation also identifies two main research challenges in advancing the proposed ICFL approach. First, constructing a complete requirement model with rich structural context requires a great deal of human effort. Second, because the requirement model and the program model describe the software at different levels of abstraction, it is not always straightforward to determine the correspondence between the type of feature dependency and the type of program-element dependency.

2. OUR APPROACH
In this section, we 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. We also 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.

2.1 Meta-Model

The meta-model of the input models that ICFL assumes is a typed directed graph. Each graph *node* represents one individual element of the model, associated with a node *type*. In our initial study, a requirement model consists of two types of graph nodes, i.e., the use-observable *functional features* and the *business objects*; a program model consists of three types of graph nodes, i.e., the *classes*, the *methods*, and the *database tables* (*db_tables*). Each node has an attribute *description*, such as the natural language description of the feature, the identifier and comments of the method.

Edges represent the directed dependencies between model elements. Each edge is associated with an edge *type*. In our initial study, a requirement model consists of two types of edges, representing the *decomposition* between features and the *read/update* dependencies between features and business objects, respectively. Similarly, a program model captures the *call* relations between methods and the *read/update* dependencies between methods and classes/db_tables. Because the requirement model and the program model describe the system at different levels of abstraction, one may need to determine the correspondence between the edge types in the two input models. For example, in our initial study, we define that the feature decomposition corresponds to the method call.

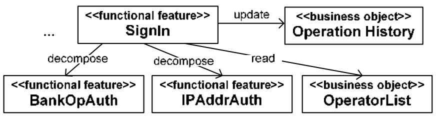

Figure 1. A partial requirement model

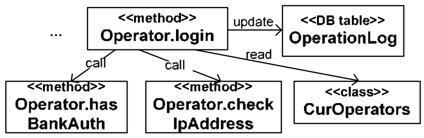

Figure 2. A partial program model

Figure 1 and Figure 2 present (partially) the requirement model and the program model from our preliminary evaluation (see Section 3) of the proposed ICFL approach with the subject system 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 the 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*.

2.2 Similarity Metrics

Let us discuss the 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.

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 [1]. 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 [7] for determining a program element's relevance to a feature. In the similar vein, given a feature f and a program element p, ICFL measure 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:

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

Structural similarity. ICFL employs an iterative feature location algorithm (see Section 2.3). Let M_i be the set of already established feature-element mappings in the iterative process. Given a feature f and a program element p, let $\{f'\}$ be the set of features that are related to f and let $\{p'\}$ be the set of program elements that are related to p, according to a given edge type t in the input models. Let $\{f''|(f'',p') \in M_i\}$ be the set of relevant features of p'. ICFL measures the structural similarity Sim_S between f and p, given the edge type t, as follows:

$$Sim_S(f,p,t) = |\{f''|(f'',p') \in M_i\} \cap \{f'\}| / |\{f''\} \cup \{f'\}|$$

The structural similarity Sim_S computes the Jaccard coefficient [6], indicating how similar two sets, $\{f''\}$ and $\{f'\}$, 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.

Overall similarity metric. Finally, given a feature f and a program element p, ICFL computes their overall similarity metric *Sim* as follows:

$$Sim(f,p) = (1-w)Sim_L(f,p) + w\sum_{t\in\{et\}} Sim_S(f,p,t) / |\{et\}|$$

where $\{et\}$ is the set of all the edge types in the input models, and w is a weight that defines the extent to which the overall similarity metric depends on the lexical similarity and on the structural similarity.

2.3 Iterative Feature Location Algorithm

Our iterative feature location algorithm is described in pseudo code in Algorithm 1. The algorithm takes as input a requirement 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. 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. 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 2.2). 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 add 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). 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). This process continues until the *threshold* is below the lower bound of similarity threshold T_{min} (line 5). Finally, ICFL returns the set Map_{F_P} of feature-element mappings (line 18).

```
1.   MapF_P ICFL(MF, MP, Tmax, Tmin, pace, w)
2.     MapF_P = Φ;
3.     CandF_P = {{f, p}| f∈MF and p∈MP};
4.     threshold = Tmax;
5.     while (threshold > Tmin) {
6.       moreMappingIdentified = false;
7.       for each (f, p) ∈ CandF_P {
8.         if(Sim(f, p) > threshold) {
9.           MapF_P = MapF_P ∪ {(f, p)};
10.          CandF_P = CandF_P \ {(f, p)};
11.          moreMappingIdentified = true;
12.        }
13.      }
14.      if(!moreMappingIdentified) {
15.        threshold = threshold – pace;
16.      }
17.    }
18.    return MapF_P;
```

Algorithm 1 The iterative feature location algorithm

Take the requirement model and the program model in Figure 1 and Figure 2 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 1 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 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 0.25. The overall similarity metric *Sim*(*SignIn*, *Operator.login*) is (1-0.8)*0.25+0.8*(1+0+1)/3=0.583.

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 their *descriptions* are not that similar lexically. Furthermore, the newly established mapping (*SignIn*, *Operator.login*) can in turn increases the structural similarity of the candidate pair (*OperatorList*, *CurOperators*) from 0 to 1, which may further push this pair above the similarity *threshold*.

Table 1. Structural similarities of (*SignIn*, *Operator.login*)

Edge Type	{f'}	{p'}	{f''}	Sim_S
decompose/call	BankOpAuth, IPAddrAuth	hasBankAuth, checkIpAddress	BankOpAuth IPAddrAuth	1
read/read	OperatorList	CurOperators	-	0
update/update	OperationHistory	OperationLog	OperationHistory	1

3. PRELIMINARY EVALUATION

We evaluate our iterative context-aware approach to feature location using a small-scale industrial system (DirectBank) from our industry partner. DirectBank, a subsystem of a financial management system developed by Wingsoft Ltd., provides bank interfacing services for cashiers. DirectBank consists of 30K lines of code, 53 classes and 414 methods.

In this evaluation, we use static program analysis (based on Eclipse JDT) to obtain the program model that captures the classes, methods, and their call/read/update dependencies. The read/update dependencies between the methods and the db_tables are extracted by statically analyzing the SQL queries in JDBC statements. The system expert of DirectBank manually identified 71 features (including business objects) and their decomposition/read/update dependencies. He also provided the description of these features. Furthermore, this system expert manually mapped these features to the relevant program elements (methods, classes, and/or db_tables) implementing them. This manually established feature-element mapping Map_{actual} serves as the "ground truth" to evaluate our ICFL approach.

We use the precision and recall metrics to evaluate the effectiveness of the proposed approach in identifying the mappings between features and their relevant program elements. Precision is the percentage of the correctly reported feature-element mappings $(Map_{F\_P} \cap Map_{actual})/Map_{F\_P}$ and recall is the percentage of feature-element mappings reported $(Map_{F\_P} \cap Map_{actual})/Map_{actual}$.

We process the *description* attribute of the features and the program elements in a similar way to other information-retrieval techniques [7], and then encode them in TF/IDF vectors associated with the corresponding features and program elements. In this evaluation, we set T_{max}=0.8, T_{min}=0.5 and *pace*=0.1. We set the weight w for computing overall similarity metric at 0.8.

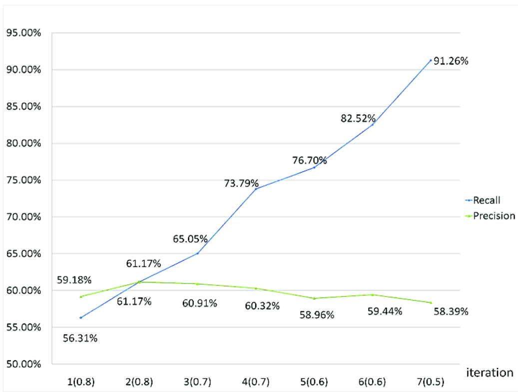

Figure 3. Precision/recall of ICFL with DirectBank

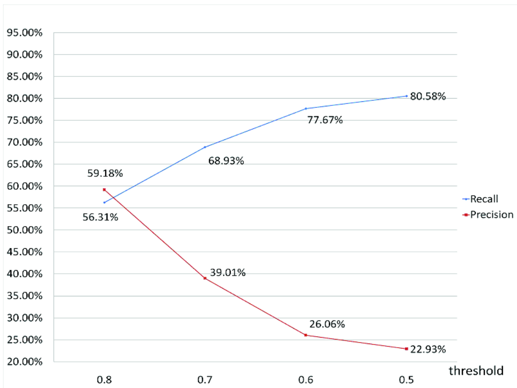

Figure 4. Precision/recall of an IR-based FL with DirectBank

The iterative feature location algorithm executes seven iterations. Figure 3 summarizes the experiment results. The horizontal axis represents the *threshold* adopted in each iteration, while the vertical axis presents the precision and recall in identifying the feature-element mappings in each iteration. 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 applied an IR-based approach to feature location [7] to the DirectBank system. Figure 4 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 description of the features and the program elements.

4. RESEARCH CHALLENGES

Our preliminary evaluation also reveals three research challenges in advancing our ICFL approach. First, our approach requires as input a requirement model and a program model. The program model can be easily obtained using static and/or dynamic program analysis techniques. But construing a complete requirement model with rich structural context information (such as use cases, feature models) may require a great deal of human effort. We believe a partial requirement model can still provide useful structural context for feature location. We are now investigating the adoption of "just enough" requirement model in our approach.

Second, because the requirement model and the program model describe the system at different levels of abstraction, it is not always straightforward to determine the correspondence between the edge (dependency) types between the two models. For example, a feature may be specialized by several other features. This feature specialization relation may be implemented using conditional compilation or runtime configuration. In such cases, one may have to understand the specific system design and implementation convention in order to define the correspondence between the edge types in the two input models. We are now investigating the impact of adopting different types of feature and program dependencies in our approach, with the goal to identify the important types of dependencies whose correspondences can be easily determined between the two input models.

Finally, we need more empirical experiences on tuning-up parameters of the iterative feature location algorithm (see Section 2.3), including T_{max}, T_{min}, *pace* and *w*. A proper initial threshold (T_{max}) is necessary to avoid the erroneously established feature-element mappings in early iterations of the algorithm execution. Such erroneous mappings can negatively affect the accuracy of structural similarity in the following iterations. A proper lower bound of similarity threshold (T_{min}) is also important so that the algorithm will not produce results with slightly better recall but much worse precision. A smaller *pace* can potentially increase the precision but it may induce higher computational cost due to more mapping iterations. Finally, a proper weight *w* should reflect the quality of the lexical description of features and program elements and the completeness of the structural context in the input models.

5. RELATED WORK

A large number of techniques for feature location have been proposed, including IR-based approaches [4], dynamic approaches [2], and hybrid approaches [5]. These approaches analyze each feature independently and perform a one-time analysis. In contrast, our approach takes into account the interdependencies (i.e., structural context) of features and program elements and performs an iterative process for feature location decisions.

Zhao et al. [7] presents a two-phase approach to feature location, which first applies an IR-based technique to identify an initial set of feature-element mappings, and then enrich the initial mappings by exploring only the program call graph. Our 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.

Lucia et al. [3] proposes an incremental IR-based feature location approach and conducts 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.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an iterative context-aware approach to automatic feature location. Taking as input a requirement 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 preliminary evaluation demonstrates that our approach can significantly increase the recall of feature location task with a slight decrease in 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.

In the future, we will focus on the following issues: (1) investigating the effectiveness of partial requirement model and the impact of different types of feature and program-element dependencies in our approach, and (2) conducting systematic empirical studies on the choice of the appropriate parameters for our iterative feature location algorithm.

7. ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under Grant Nos. 60703092 and 90818009.

8. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press and New York: Addison-Wesley, 1999.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE T Softw. Eng.* 29, 3 (2003), 210–224.

[3] A. Lucia, R. Oliveto, and G. Tortora. IR-Based Traceability Recovery Processes: An Empirical Comparison of One-Shot and Incremental Processes. In *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ASE'08, 39-48.

[4] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. of the 25rd Int. Conf. on Softw. Eng.*, ICSE'03, 125-135.

[5] D. Poshyvanyk, Y.G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE T Softw. Eng.* 33, 6 (2007), 420–432.

[6] P.N. Tan, M. Steinbach and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[7] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. *ACM T Softw. Eng. Meth.* 15, 2 (2006), 195–226.