A Top-down Feature Mining Framework for Software Product Line

Yutian Tang, and Hareton Leung
Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong SAR, China
{*csytang, cshleung*}*@comp.polyu.edu.hk*

Keywords: Variability, feature mining, concept location, top-down framework, software product line.

Abstract: Software product line engineering is regarded as a promising approach to generate tailored software products by referencing shared software artefacts. However, converting software legacy into a product line is extremely difficult, given the complexity, risk of the task and insufficient tool support. To cope with this, in this paper, we proposed a top-down feature-mining framework to facilitate developers extracting code fragments for features concerned. Our work aims to fulfill the following targets: (1) identify features at a fine granularity, (2) locate code fragments for concerned feature hierarchically and consistently, and (3) combine program analysis techniques and feature location strategies to improve mining performance. From our preliminary case studies, the top-down framework can effectively locate features and performs as good as Christians approach and performs better than the topology feature location approach.

1 INTRODUCTION

Software product line engineering (SPLE), a promising approach in generating software product at a low cost, is broadly applied and adopted in software production nowadays (Kang et al., 2009). SPLE provides software products in a large-scale with systematically reuse of software assets, including documents, source code, design and so forth (Chen et al., 2009). Moreover, SPLE enables tailoring software products to satisfy various customers by providing products with feature diversity.

Features, designed and declared in the specification, are used to represent functions and characteristics of the system. Conceptually, features can be classified into two categories including common and distinguished features. Specifically, common features are implemented in all products within the software product line (SPL). On the contrary, distinguished features denote variants in SPLE, which are employed to generate tailored software products for customers (Benavides et al., 2010). For instance, a banking product line provides common features including creating accounts, deposit, and withdrawal from accounts as well as some distinguished features such as multi-language support and exchange calculation.

Despite SPL supports customized software products at a low cost and optimized time to market, the adoption rate of SPL has been low, since suppliers have to take the risk of converting software legacy into SPL. Meanwhile, the complexity and the cost of this task will also prevent suppliers to adopt SPL as their preferred approach to generate software products. Software product line could be built based on software legacy by extracting and refactoring features, which could save the cost at the preliminary stage, at which variants are designed and implemented. Migrating software legacy to SPL normally includes the following procedures: extracting features from legacy; transferring extracted features into a feature model; and building product line architecture under the feature guide (Laguna and Crespo, 2013).

To partially resolve the issues unsolved in migrating legacy to product line and enhance the performance in feature mining, in this paper, our study will: (1) provide a strategy to mine source fragments from software legacy in fine granularity; and (2) reduce the cost and risk when migrating software legacy to SPL and assist developer to locate specific feature efficiently.

The rest of this paper is organized as follows: in Section II, our top-down feature mining framework along with supporting recommendation strategies will be introduced. Case study and experimental results will be presented in Section III. Section IV reviews the techniques related to feature mining. Conclusion will be given in the last section.

2 PROPOSED FRAMEWORK AND METHODOLOGY

Our vision is to recommend code fragments to developers to support learning and analyzing the code legacy. As described in (Kastner et al., 2014), the skeleton of a feature mining process should include following steps: (1) a domain expert describes the features and internal relationship, (2) automatically or manually select the initial seeds to start feature searching, (3) iteratively expand identified code to include more code fragments that have not been assigned to features temporally, and (4) developers could extract and rewrite the code segments as variants. Unfortunately, it is unrealistic to provide a fully automatic approach to extract all code fragments attached to the feature concerned given the complexity and risk of the task. To follow the general procedure of feature mining, in the coming subsections, we will introduce our top-down framework by covering the underlying model, extraction of relationship between programming elements, recommendation mechanism and concrete process of our top-down approach.

2.1 Fundamental Elements

The fundamental elements of a program are programming elements and relationships among them. Technically, a program can be represented by a standard graph with nodes to represent programming elements and links to depict relations. Specifically, programming elements (nodes), in fine granularity, could be fields, methods, statements, expressions, and local variables. Meanwhile, relations between programming elements describe the structure of the program and dependencies between elements as well. Particularly, in Java, we could extract programming elements using Java Development Tookit (JDT) API, which is also the kernel compilation mechanism in Eclipse. Relationships between programming elements are normally embedded in keywords defined in programing language under grammar rule and could be employed to depict the structural information of the code base. Specifically, in this paper, we adopt AST (Abstract Syntax Tree) nodes to represent programming elements. AST is an abstract representation of the program by categorizing programming elements by types, for instance, *IfStatement*, *ForStatement*, and *VariableDeclarationExpression*.

2.2 Relation Extraction

Relations between programming elements need to be normalized to be displayed directly and quantitatively. All relations can be classified into control flow and data flow. In this paper, we extract fundamental programming elements by JDT and transfer keywords under grammar rules of specific programming languages into the control flow and data flow respectively. Specifically, we divide the data flow relation into two subsets by granularity: statement level and function level. Statement level depicts how data are transferred within a method. By contrary, function level data flow, also known as call flow, provides method invocation information and guides execution path from one method to another.

2.2.1 Control Flow

Control flow relation is normally represented by a control flow graph, which describes how the program is constructed and organized. We adopt the control flow model described in (Sderberg et al., 2013), which constructs the control flow graph based on abstract syntax tree (AST) of a program. According to (Sderberg et al., 2013), control flow nodes could be classified into three types, including **non-directing**, **internal flow**, and **abruptly completing**, according to different types of execution control. Concretely, **non-directing** node decides the next node by the context; *varAccess* node, which represents variable access, is an instance of this type. In practice, a statement node could contain various sub AST nodes. For instance, in *ForStmt* (for statement), *ForInit*, Expression, *ForUpdate* and forbody statements are embedded in *ForStmt*. **Internal flow** node directs the control flow inside the node and traverses all AST nodes within it. For the *ForStmt* case, the control flow will go through *ForInit*, Expression, forbody, and *ForUpdate* until the exit criteria is met. In addition, *WhileStmt* and *IfStmt* are also examples in this category. **Abruptly completing** node allows the program goes to a specific location, which is outside the scope of current block; examples include *Break* and *Return*.

2.2.2 Data Flow(Statement Level)

Data flow depicts how data are transferred, accessed and stored inside the program and is mainly built on the control flow graph. Technically, there are two types of data flow analysis (Sderberg et al., 2013): **liveness** analysis, which detects the state of variable at a certain point within the program, and **reaching definition** analysis, which identifies the destination that a definition can reach. Particularly, **liveness** checking is applied for dead assignment checking and reaching definition analysis mainly detects *use-define* and *define-use* chain. To simplify this analysis, we only detect and build *use-define* and *define-use* chains

(both intra-procedural and inter-procedural); for example, we build the connection between variable access and variable declaration, rather than checking for any dead assignment. This aligns with our main goal of locating feature code relate to a certain feature rather than finding potential program defect, which is the key application of dead assignment checking.

2.2.3 Call Flow(Function Level)

Call graph (Rountev et al., 2004) supports analysis of calling relationships between methods, and displays the potential behavior of the program. In call graph analysis, two types of graphs (static and dynamic), could be applied. Static call graph analysis merely relies on structural information represented by program. On the other hand, dynamic analysis also takes runtime information into account. In this paper, we adopt static call graph to detect method invocation relationships. In static analysis, we extract following three relationships: (1) method m_i calls method m_j;(2) call site i_k in method m_i invokes method m_j; and (3) call site i_k in method m_i invokes method m_j on an instanceof X, where X is the identifier of class. By extracting call relations mentioned above,the call relation at fine granularity could be derived.

2.3 Recommendation Mechanism

To support feature-mining, we provide a set of recommendation strategies, which will assist locating features precisely. Concretely, for a programming element, there is a neighbor-set, which contains all programming elements, which may have control relation and data (reference) relation. Nevertheless, the connections between a programming element and its neighbors could be strong and weak, which means this element and its neighbor elements could and could not belong to the same feature. To rank all programming elements inside the neighbor-set, we use text comparison, distance-based analysis, centrality-based analysis, and topology approach to determine their relevancy.

2.3.1 Text Comparison

Features could be extracted from textural aspect when treating a source code file as a set of tokens. Thereby, similarity between two programming elements can be measured from the textural aspect. Using text comparison enables examination of potential connection between the declaration of method, fields, variable, and class. Text comparison is not restricted to different programming elements, as it can identify the importance of various token and feature. Particularly, for a feature, we identify a list of tokens (feature descriptor) serving as a descriptor of this feature. When a new code fragment is classified into this feature, the featured tokens embedded in it will be added to this feature list automatically. Within this feature list, all tokens are ranked based on a weight value according to Christians approach (Kastner et al., 2014);that is,a token is ranked by counting relative occurrences of substring in the current feature list after subtracting occurrences of list outside this feature. That is:

$$score_{TR}(e, extent(f), exclusion(f)) = \sum_{t \in tokenized(e)} (F(extent(f),t) - F(exclusion(f),t)) \cdot \rho(t) \quad (1)$$

,where $extent(f)$ represents the feature token list under consideration, $exclusion(f)$ contains all programming elements currently not annotated to feature f, e denotes a programming element, and function F gives the frequency of token t in the feature list.

2.3.2 Distance-based Analysis

In a broader sense, clustering can be used for feature mining, when the goal is to explore and extract all code fragments attached to the feature concerned. Specifically, each cluster could be adapted to represent a unique feature and all nodes within the cluster are programming elements belonging to this feature. As mentioned previously, a program could be interpreted as a graph with programming elements (nodes) and relationships (links). In graph clustering, distance-based clustering algorithms are widely adopted, of which K-medoid (Dhillon et al., 2005) is a representative one. For feature mining purpose, we modified the original algorithm by considering following scenarios to classify un-annotated programming elements: (1) for a single programming element e, if it merely connects to an annotated feature cluster c_1, we compute its distance by: $score_{DA}(e, c_1) = \sum_{t \in neighbors(c_1)} D(e,t)/neighbors(e)$,to denotes the possibility that it belongs to a feature cluster and t represents e's neighbor node inside the cluster c_1. For example, in Figure 1, node 11 is linked with node 12 only.(2) If a programming element is connected to various elements, which belongs to different clusters as shown in Figure 2, then for any two neighboring elements, we compute the distance as: $score_{DA}(e, c_1, c_2) = (\sum_{t_1 \in neighbors(c_1)} D(e,t_1) - \sum_{t_2 \in neighbors(c_2)} D(e,t_2))/neighbors(e)$, where e is the programming element currently under consideration, and $\sum_{t_i \in neighbors(c_i)} D(e,t_i)$ represents the total distance between element e and all its neigbhors in cluster i. Therefore,there is a negative correlation between $score_{DA}$ and the rank of the programming

element. To simplify, we adjust the formula to:

$$score_{DA}(e, extent(f), exclusion(f)) = \\ (\sum_{t_1 \in neighbors(exclusion(f))} D(e, t_1) \\ - \sum_{t_2 \in neighbors(extent(f))} D(e, t_2)) / \\ \max(|neighbors(extent(f))|, |neighbors(exclusion(f))|) \tag{2}$$

where $extent(f)$ represents proramming elements inside the feature f, and $exclusion(f)$ shows elements currently not annotated to feature f.We adjust the original formula by changing the order of $\sum_{t_1 \in neighbors(exclusion(f))} D(e, t_1)$ and $\sum_{t_2 \in neighbors(extent(f))} D(e, t_2)$ to guarantee the similarity is positively correlated with $score_{DA}$, which means the higher is the score, the larger the distance to exclusion of feaeture f. That is, a higher score indicates that a higher possibility of recommending this programming element to the feature f.

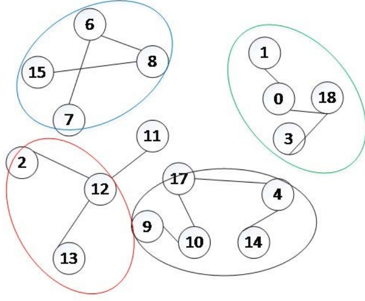

Figure 1: Example of single connection.

2.3.3 Centrality-based Analysis

Girvan-Newman algorithm (Newman and Girvan, 2004) is a graph-based community detection approach, which implements graph partitioning according to *betweenness centrality*. The concept *betweenness centrality* describes a property of the network as well as the shortest paths between nodes, and it is normally defined as: for edge $e \in E, B(e) = \sum_{u,v \in V} \frac{g_e(u,v)}{g(u,v)}$, where $g(u,v)$ represents the number of paths from u to v and $g_e(u,v)$ represents the number of paths from u to v via e. Their algorithm ranks all edges by betweennes and removes the highest one iteratively. Girvan-Newman has been shown to perform well for graph clustering in practice (Figueiredo et al., 2008). To fit our particular need in software product line feature mining, we compute priority based on the extent and exclusion of the feature.

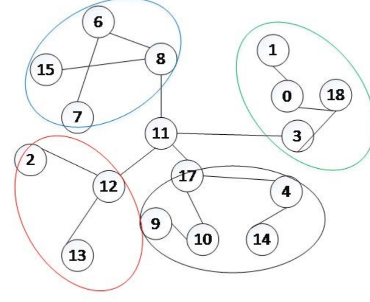

Figure 2: Example of multiple connections.

$$score_{CA}(e, extent(f), exclusion(f)) = \sum_{i \in extent(f), j \in exclusion(f), v \in V} \frac{g_e(i,v) - g_e(j,v)}{g_e(i,v) + g_e(j,v)} \tag{3}$$

2.3.4 Topology Analysis

Topology approach provides recommendations by referencing the topology structure of a graph. Here, we adopt Robillards topology approach (Robillard, 2008) to identify interested programming elements. To rank all potential programming elements, it employs *specificity* and *reinforcement* metrics. *Specificity* describes the case that if one programming element only refers to a single element, it will rank higher than others that refer to many elements. In contrast to *specificity*, the underlying intuition in reinforcement is that an element, which is referred by many annotated elements, should be ranked higher. Unfortunately, the original algorithm described in (Robillard, 2008) is merely designed for method and field instead of programming elements at the fine granularity including statements, local variable and so forth. Therefore, we adjusted the original algorithm to the product-line settings by applying it to the fine granularity programming elements. To rank the priority of a programming element among all candidates, we employed the following formula:

$$score_{TA}(e, f) = topology(e, extent(f)) - \\ topology(e, exclusion(f)) \tag{4}$$

where topology is defined as

$$topology(e, X) = \frac{1 + |forward(e) \cap X|}{|forward(e)|} \cdot \frac{|backward(e) \cap X|}{|backward(e)|} \tag{5}$$

Forward represents a set of programming elements, which are the target nodes of relation links.By contrast, *backward* refers to a group of elements, which are the starting points of relation links. For instance, in Figure 3, the forward set of node d is $forward_d = \{a, b, e\}$ and the backward set of node b is $backward_b = \{d\}$.

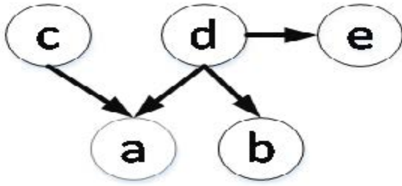

Figure 3: Example of backward and forward relation in a graph.

2.3.5 Integrating all approaches

To provide an overall recommendation strategy, we adopt Robillards (Robillard, 2008) operator to merge all recommendation approaches mentioned above. In this approach, the formula $x \uplus y = x + y - x \cdot y$ is utilized to combine two priority scores x and y to rank the potential elements. Specifically, our integrated strategy will consider all aspects and methods discussed above using the $score_* = score_{TR} \uplus score_{DA} \uplus score_{CA} \uplus score_{TA}$.

2.4 Top-down Framework

In previous sections, we presented the strategy to extract potentially related programming elements for a give programming element and four approaches to rank all potential programming elements from various aspects. Here potential programming elements mean a set of programming elements, which connect with a give element under a certain relation, which could be data (call) or control. Next, we will introduce a top-down framework by presenting seeds selection and the key steps of our framework respectively. Our process is named top-down because it locates the feature from coarse granularity to fine granularity. That is, firstly it identifies the class and method, which may be related to the feature. Then, it locates the specific statements inside the method.

Seeds selection. To launch the top-down framework, seeds should be determined by developers or domain experts initially. Although they may not be familiar with the whole implementation, developers can nevertheless propose some start up points. Specifically, in our method, all initial seed recommended are methods, which are *MethodDeclaration* nodes in java AST.

Top-down process. Our framework first locates the class and method that may be related to feature. Then, it looks inside the method and inspects associated code segments for the feature concerned. The top-down feature mining process contains four steps as illustrated in Figure 4:
① Call relation, data flow, and control relation; ② Data and control relation (Re.Stmts) and call relation (Re.Method); ③ Replace; ④ Extract and rewrite.

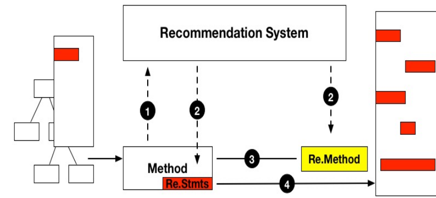

Figure 4: Top-down recommendation procedure.

- **STEP1**:Initially, a domain expert or developer will be required to select the seed (a single or a set of methods) for feature concerned to launch the mining process. This procedure is essential and the seed should be carefully chosen to ensure that it could be applied to represent the feature;
- **STEP2**:If it is the first iteration, the start-up method(s) should be the seed(s) selected by developers or domain experts. The start-up method(s) serves as input to the recommendation system, which will return two groups of programming elements. The first group includes recommended methods and the second group programming elements within the start-up method(s). These programming elements are considered to be part of code that implements the feature. In this way, the goal of fine granularity mining is achieved, since statement-level programming elements are inspected to be contained in the feature concerned;
- **STEP3**:The returns of STEP2 are recommended method and statements based on the start-up method. If the returned method is not null, it will be redirect to STEP2 and serve as the start-up method. For instance, first we select method BankAccount as the start-up method in STEP2 and the recommendation system provides the method *getBankName* as result. Then, in STEP3, the method *getBankName* will turn back to STEP2 and serve as the start-up method;
- **STEP4**:Eventually, code fragments related to the feature concerned will be extracted.

Next, we will demonstrate the recommendation mechanism using the example shown in Figure 5:

- To extract code fragments related to feature *locking*, developers first annotate the method *lock* as a seed. After that, the call graph and control chain graph are employed to detect internal and external relations of this method. That is, method *push*, *pop* and class *lock* are taken into further consideration using the recommendation mechanism (STEP 1).

- Next, the control and data flow are utilized to detect reference and potential branches, which may jump to code fragments that belong to other features. For instance, assuming that feature *Lock* is the feature cared; the developer/domain expert might first pinpoint method *lock* in class *stack*. Moving on, methods which invoke *lock* (*push* and *pop*) or invoked by *lock* (*acquire*) will be recommended by the framework as a result of STEP 2. Method *push*, *pop* and *acquire* will serve as input of STEP 3, and then method *unlock* will be suggested by our system.

```
class stack {
    int size = 0;
    Object[] elementData = new Object[maxSize];
    boolean transactionsEnabled = true;

    void push(Object o) {
        Lock l = lock();
        elementData[size++] = 0;
        unlock(l);
    }
    Object pop() {
        Lock l = lock();
        Object r = elementData[--size];
        unlock(l);
        return r;
    }
    Lock lock(){
        if(!transactionsEnabled) return null;
        return Lock.acquire();
    }
    void unlock(Lock lock) {/*...*/}
    String getLockVersion() {return "1.0";}
}
class Lock {/*...*/}
```

Figure 5: Stack example shows feature locking (related code are all highlighted).

Stopping criteria. We next introduce the stopping criteria of our top-down framework, which defines the condition that the algorithm for a single programming element (STEP 1 and 2 in Figure 4) will halt. When one of following criteria is met, the recommendation for current programming element will stop:

- All (data or control) neighbors of the current programming element are annotated by other features: when all data and control neighbors of the current programming element are annotated to some features, the mining procedure for this specific programming element will stop.
- All neighbors of the current programming elements are computed by using the recommendation system. A threshold is set to select programming elements from all potential pools. That is, if and only if the score computed is greater than the threshold, then it will be annotated to this feature. Otherwise, it will be un-annotated and extra analysis is done to decide whether it should be annotated by other features.

When there is no fresh programming element recommended by the top-down framework, the mining process for current feature will stop. Top-down framework will consider following attributes of product line engineering when mining feature from legacy:

- **Consistent feature mining**. To resolve the specific need in product line context, if a feature is extracted and all code fragment related to this feature should be explored and located as well, since in further step of product line constructing, variants will be compiled and constructed to implement product line variability. Top-down framework tries to inspect relative code segment iteratively (in STEP 2 and 3) to fulfill the consistent mining need.
- **Binary mapping**.For a specific programming element at fine granularity (could be a statement or a variable), it should be classified to a feature. That is, the relation between a feature and all code segments implemented it, is a belong-to relation. Binary mapping means there are two possible cases to denote the relation between a feature and a programming element, which are include and exclude. To deal with this, we proposed several approaches to depict the closeness between a feature and a programming element, with the computed value supports the binary decision as described in section 2.

3 EVALUATIONS

3.1 Case Study

To reduce bias and for comparison purpose, we select practical products that have been widely used for our study. We select following systems including *Prevayler*, *MobileMedia*, and *Sudoku*. To exclude the bias in selecting features, all main features within these systems are selected. In this paper, we concentrate on feature mining, and detecting feature interaction is beyond the scope of this paper. Therefore, all features are equally treated.

Prevayler. Prevayler is an open source object persistence library for Java. It is a Prevalent System design pattern for keeping live in memory and transaction during system recovery. This project is well investigated in (Kastner et al., 2014; Valente et al., 2012) . We adopted the same Prevayler version[1]researched by de Oliveira at the University of Minas Gerais, with five features, including *Censor*, *Gzip*, *Monitor*, *Replication*, and *Snapshot*.

MobileMedia. MobileMedia is a medium size product line originally designed by University of

[1]See http://spl2go.cs.ovgu.de/projects/35.

Lancester to provide various operations including manipulate photo, music and video on mobile device on Java ME platform. Specifically, it offers six features including *Photo*, *Controller*, *Count Views*, *Persistence*, *Favorites*, and *Exception Handling*. MobileMedia[2] is a mature product line that has been analyzed and investigated by many studies (Figueiredo et al., 2008).

Sudoku. Sudoku is a simple java puzzle game designed by University of Passau, containing 1975 lines of code within 26 files. It includes five features: *Variable Size*, *Generator*, *Undo*, *Solver*, and *States*. Considering the structure described in (Apel et al., 2009), this product is designed with a composition-based approach.

3.2 Quantitative Evaluation

To assess our framework for feature mining from software legacy, we implement a tool named JFeTkit (Java Feature Mining Tookit) to extract featured code from the software legacy. JFeTkit is a compound system, which uses several existing software analysis libraries, including BCEL (Byte Code Engineering Library), Crystal[3] analysis framework and JDT (Java Development Tookit). JFeTkit collects the information generated using these third-party APIs and annotates software code legacy using our framework. To start, the recommendation system provides a single programming element each time. When the algorithm stops, a set of programming elements are annotated to the features concerned. Next, code fragments are extracted and rewritten in terms of feature.

To evaluate the performance of our recommendation mechanism quantitatively, metrics *recall* and *precision* are adopted. *Recall* is the number of correct results over the number of total results returned. That is, the percentage of feature code detected and annotated by the algorithm comparing to all code within this feature. *Precision* describes the number of correct recommendations comparing to all inspected code. Here, we adopted the same quantitative functions as defined in (Kastner et al., 2014) and the metrics are calculated based on LOC (line of code):

$$\begin{aligned} Precision &= \frac{Correct\ recommendation}{All\ code\ inspection\ when\ stop}, \\ Recall &= \frac{Lines\ of\ code\ annotated\ when\ stop}{Lines\ of\ code\ annotated\ in\ original} \end{aligned} \tag{6}$$

[2]Version 6, in 2009 and the feature name used in presented in (Figueiredo et al., 2008).

[3]Crystal analysis framework: https://code.google.com/p/crystalsaf/.

3.3 Mining Result

Table 1 gives the line of recommended code for each feature, precision and recall results along with the basic information for studied systems. For all three systems, our framework can reach a recall of 95% on average along with a precision of 44%. Recall denotes the degree at which our framework can extract the source code for specific feature concerned. The recall results indicate our framework could detect and extract almost the entire code fragment for the feature concerned. That is, for a single feature, the method could extract 95% of all programming segments, which should be annotated to this feature. According to the precision result, unfortunately, the low score means our framework sometimes provides some unnecessary code segments for developers. However, by following the annotated code fragments, developers will have a general understanding and insight into the code legacy instead of searching repository for code segments that implement a certain feature. Thereby, developers could find most code fragments for the feature concerned, although additional effort is required to improve the precision and extract the complete code fragments.

We compared the performance of our top-down framework with Christians semiautomatic variability mining approach(Kastner et al., 2014) and the topology approach as shown in Figure 6 and 7. Here, the values on horizontal axis represent ten features in comparison. The underlying reasons to choose Christians variability mining approach for comparison are follows: (1) both approaches are designed for software product line use instead of traditional software systems, since both consider fine granularity instead of coarse level. Differently, Christians approach can be used to compare the similarity of fine level programming elements using strategy pool, which contains text comparison, type checking, and topology approach. (2) Secondly, we adopted the same performance assessment metrics of recall and precision, also the same evaluation procedure as described in 3.2. We found that the top-down framework performs as well as the semiautomatic variability mining approach and shows strength comparing to topology approach. As demonstrated in Figure.6 and 7, our feature mining approach performs almost as good as Christians approach with small improvement in recall.

Additionally, we compare our feature-mining framework with the topology approach, which is a traditional feature location strategy. Note that the topology approach is one of recommendation strategies used in our framework. It was originally de-

signed for coarse granularity, which could only locate methods and fields instead of elements at the statement level. However, it has not been adapted for feature mining in software product line. We compare our merged recommendation system with topology for the following reasons: (1) among all approaches within our framework, only the topology approach is designed for feature location, and (2) As, the topology approach is a sub-method embed in our approach, it is natural to provide comparison with this approach. Although the original topology method is not fully adapt to product-line context, we adjust it to serve the fine granularity need. Consequently, the topology approach could reach a recall of 79% and precision of 37% on average as shown in Figure. 6 and 7. Therefore, our approach performs better than the topology approach in both recall and precision.

We found following scenarios that may influence the performance of our approach:

- **Seed selection**. The performance of our framework is highly influenced by the selection of initial seeds. A single seed sometimes might not be sufficient to represent a feature, and the seed selected may not be strongly connected to a particular feature, that is, the seed may also contain code segments belonging to other features. These might be the reasons that lead to the bad performance of Sample 7 shown in Figure 6 and 7.
- **Paralleled feature selection**. Comparing to extracting featured code one after one, if domain experts or developers select multiple seeds for each feature concerned, the resulting precision will be higher, since interaction among features will be considered together. For instance, considering a feature F in a Switch-Case statement, if there is one sub-case branch annotated to feature T, then local variables only used in this sub-branch should be annotated to T.

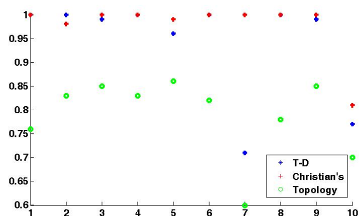

Figure 6: Recall performance comparison

In any case, the top-down approach shows advantages in efficiency, since, in essence, it is a searching approach. To improve its performance, following steps could be adopted: (1) select multiple seeds instead of a single one, and (2) let multiple features launched simultaneously to enhance the feature interaction.

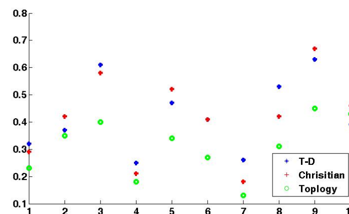

Figure 7: Precision performance comparison

Table 1: Feature Mining Result

Project	Feature	LOC	Recall	Prec.
MobileM.	Persist.	748	100%	42%
	Favorite	88	100%	44%
	CountV.	96	98%	26%
	Control.	423	100%	68%
	Photo	185	91%	49%
	Exce. Hand.	434	99%	61%
Sudoku	Vari.Size	44	100%	32%
	Generator	172	100%	37%
	Solver	445	99%	61%
	Undo	39	100%	25%
	State	171	96%	47%
Prevalyer	Censor	105	100%	41%
	Gzip	161	71%	76%
	Monitor	240	100%	53%
	Replica.	1487	99%	63%
	Snapshot	263	77%	39%

3.4 Threats to Validity

To assess the experimental setting and procedure, and ensure that experiments have been designed properly, we consider the procedure to avoid bias introduced by inappropriate settings. Specifically, three key aspects of threats, including construct, internal and external validity, will be discussed in this section. Construct validity is the degree to which the variables measure the concept purport to measure. Internal validity is the degree, which reflects a causal relation between the presumed process and expected result. External validity is the degree to which experimental results can be generalized to a larger population. Construct validity and internal validity are highly dependent on the projects selected for the case study. We studied the selected software projects carefully to guarantee all chosen cases are mature and suitable, which means that features can be clearly defined as investigated in previous research. Furthermore, the recall and precision metrics are established upon LOC to count code

lines instead of function points to measure the performance of our framework in a fine granularity. Due to the tool we use, there may be some inaccuracy in the metrics result. Another issue of internal validity is that sometimes a single line shared or annotated by more than one feature might rise the risk of introducing interference. Notice that this case will only occur when launch the framework to locate feature sequentially, since when there are more than one feature inspecting the same programming element, the closeness between features and the programming element will be computed to decide belongings. For external validity, since the object of experiments is to verify and test the performance of our framework, the conclusions drawn are simply based on the cases selected and not attempt to extend to all systems. Although the scope of applying our conclusion is limited, the selection of case projects is reasonable, since they are from research and industrial settings.

4 RELATED WORK

Feature mining is highly associated with feature location, reengineering, feature identification and other relevant fields(Laguna and Crespo, 2013). The primary task of feature mining is to map features to the related source code, which is often scattered all over the system. Typically, techniques involved mainly include static, dynamic and textural analysis. Most approaches proposed are targeting conventional systems instead of product lines. Whereas these approaches are not principally designed for the SPL purpose, enhancement to these techniques could be applied to meet the specific circumstances in SPL. Due to space restriction, this section briefly introduces some representative approaches instead of providing a comprehensive list of relative work. Comprehensive reviews can be found in (Dit et al., 2013; Galster et al., 2013; Laguna and Crespo, 2013).

4.1 Static Analysis

In static analysis, the underlying structural relations are extracted including structural control or data flow without requiring execution information. Generally, static analysis concentrates on mining program elements related to the initial set, which is chosen by developers, based on a dependency graph and a series of software artifacts manually or automatically(Dit et al., 2013). Over the passing years, a vast amount of research work has been conducted in static analysis from various aspects. Specifically, Chen provided an static analysis approach based on Abstract System Dependence Graph (ASDG) to locate the code fragments(Chen et al., 2009). Benefit from Robillard and Murphys work(Robillard and Murphy, 2007), a feature can be displayed in an abstract way, which allows creating and saving the mappings between features and code. Later, Robillard(Robillard, 2008) described the topology approach to extract programming elements based on the initial set, which mainly serves as input set to mine all relevant elements, with potential elements ranked by two different strategies named specificity and reinforcement respectively. In addition, searching strategy is another feasible direction in the category of static analysis. For instance, Saul et al. presented an approach using a random walk algorithm with a given starting point as input (Saul et al., 2007).

4.2 Dynamic Analysis

On the contrary, execution information is collected for analysis during runtime in dynamic analysis, which is principally belonging to program comprehension. Due to its long history, a vast amount of research works has been conducted. Initially, dynamic analysis mainly supports debugging, testing and profiling, since providing related source code at hand is crucial for that kind of analysis. Focusing on feature location, Wilde and Scullys work (Wilde and Scully, 1995) provides a software reconnaissance tool basically relying on dynamic information. The extension of (Wilde and Scully, 1995) is called Dynamic Feature Traces (DFT), provided by Eisenberg (Eisenberg and De Volder,). This approach enables test scenarios used to examine the feature to collect execution information (Cornelissen et al., 2009). Other research works include Wong et al.(Wong et al., 2000), Antoniol et al.(Antoniol and Gueheneuc, 2006) and Poshyvanyk et al.(Poshyvanyk et al., 2007).

4.3 Textual Analysis

Some useful information embedded in textual information may imply the related feature, since identifiers and comments are highly related to the feature that code fragments are attached. This process includes information retrieval (Cleary et al., 2009; Poshyvanyk et al., 2007) and natural language processing (Hill et al.,). Besides that, some code engines are employed to search for code. For example, LSI (Mar,), SNIFF(McMillan et al.,), and $FLAT^3$ (Savage et al., 2010) are also well investigated to provide competitive mining results.

4.4 Tool

In this section, tools which support feature-mining process will be presented based on the types of technique. Rather than providing a comprehensive survey, we just list some representative tools as shown in Table 2, where *Dyn.* represents dynamic and *Text.* denotes textural. Detailed description can be found in (Dit et al., 2013).

Table 2: Feature location tools

Name	Type	Attributes
Ripples (Chen et al., 2009)	Static	ASDG, impact analysis
Sude (Warr and Robillard, 2007)	Static	Topology analysis
TraceG. (Lukoit et al.,)	Dyn.	Emphasize first time call
STRADA (Egyed et al., 2007)	Dyn.	Logical constraints to exclude unrelated code
GES (Poshyvanyk et al., 2006)	Text.	Unobtrusively re-indexes search spaces
IRiSS (Poshyvanyk et al., 2005)	Text.	Sort the results by granularity

In summary, approaches presented mainly work at the coarse granularity, which are not suitable for extracting fine level statement for software product line. The reason fine granularity is critical is that after feature location, code segments should be extracted, rewritten and organized to variants. Later, these variants will serve as basic infrastructures and components of product line. Thereby, current approaches should be adjusted to meet the specific need in the software product line context. Our approach is like a compound approach of feature location techniques; particularly, we also apply clustering techniques to improve feature mining. Comparing to original approaches in feature location, the top-down framework shows advantages in the following aspects:

- It is designed for fine granularity instead of coarse granularity for the particular circumstance in SPL, since featured code should be extracted and refactored to variants. Approaches listed in related work are mainly designed for conventional software instead of product line; thus they are not very effective for feature mining work in software product line.
- A single strategy could provide a competitive mining result in term of feature mining, but it might not have a comprehensive analysis on the system. Therefore, some meaningful information, which indicate relations among programming elements and features may somewhat omitted. To cope with this issue, top-down framework combines multiple methods to provide a more precise recommendation.

5 CONCLUSIONS

Software product line engineering has received great attention, as it offers software products with customized features and opportunity for reuse. To reduce the risk, complexity and cost of adopting software product line, feature mining helps to extract useful features from software legacy code. In this paper, we demonstrate a top-down feature-mining framework, which binds feature location strategies and program analysis techniques. In the case studies with 3 systems, we demonstrate that our top-down framework performs well in feature location.

For future work, we will initially migrate more existing feature location strategies to our recommendation system to improve its performance. One feasible direction is to introduce dynamic program analysis for extracting the relationship between programming elements. Furthermore, to reduce potential impact introduced by incorrect seeds selection, some natural language processing algorithms could be considered to improve the seed selection process.

REFERENCES

Antoniol, G. and Gueheneuc, Y. G. (2006). Feature identification: An epidemiological metaphor. *Software Engineering, IEEE Transactions on*, 32(9):627–641.

Apel, S., Kastner, C., and Lengauer, C. (2009). Featurehouse: Language- independent, automated software composition.

Benavides, D., Segura, S., and Ruiz-Corts, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

Chen, L., Babar, M. A., and Ali, N. (2009). Variability management in software product lines: a systematic review.

Cleary, B., Exton, C., Buckley, J., and English, M. (2009). An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702.

Dhillon, I., Guan, Y., and Kulis, B. (2005). A fast kernel-based multilevel algorithm for graph clustering. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 629–634. ACM.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

Egyed, A., Binder, G., and Grunbacher, P. (2007). Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 41–42. IEEE Computer Society.

Eisenberg, A. D. and De Volder, K. Dynamic feature traces: Finding features in unfamiliar code. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 337–346. IEEE.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., and Dantas, F. (2008). Evolving software product lines with aspects. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 261–270. IEEE.

Galster, M., Weyns, D., Tofan, D., Michalik, B., and Avgeriou, P. (2013). Variability in software systems-a systematic literature review.

Hill, E., Pollock, L., and Vijay-Shanker, K. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242. IEEE Computer Society.

Kang, K. C., Sugumaran, V., and Park, S. (2009). *Applied software product line engineering*. CRC press.

Kastner, C., Dreiling, A., and Ostermann, K. (2014). Variability mining: Consistent semiautomatic detection of product-line features.

Laguna, M. A. and Crespo, Y. (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, 78(8):1010–1034.

Lukoit, K., Wilde, N., Stowell, S., and Hennessey, T. Tracegraph: Immediate visual location of software features. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 33–39. IEEE.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120. IEEE.

Newman, M. E. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113.

Poshyvanyk, D., Gueheneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions on*, 33(6):420–432.

Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A. (2005). Iriss-a source code exploration tool. In *ICSM (Industrial and Tool Volume)*, pages 69–72.

Poshyvanyk, D., Petrenko, M., Marcus, A., Xie, X., and Liu, D. (2006). Source code exploration with google. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 334–338. IEEE.

Robillard, M. P. (2008). Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):18.

Robillard, M. P. and Murphy, G. C. (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3.

Rountev, A., Kagan, S., and Gibas, M. (2004). Static and dynamic analysis of call chains in java. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 1–11. ACM.

Saul, Z. M., Filkov, V., Devanbu, P., and Bird, C. (2007). Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 15–24. ACM.

Savage, T., Revelle, M., and Poshyvanyk, D. (2010). Flat 3: feature location and textual tracing tool. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 255–258. ACM.

Sderberg, E., Ekman, T., Hedin, G., and Magnusson, E. (2013). Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827.

Valente, M. T., Borges, V., and Passos, L. (2012). A semi-automatic approach for extracting software product lines. *Software Engineering, IEEE Transactions on*, 38(4):737–754.

Warr, F. W. and Robillard, M. P. (2007). Suade: Topology-based searches for software investigation. In *Proceedings of the 29th international conference on Software Engineering*, pages 780–783. IEEE Computer Society.

Wilde, N. and Scully, M. C. (1995). Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62.

Wong, W. E., Gokhale, S. S., and Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98.