StiCProb: A Novel Feature Mining Approach using Conditional Probability

Yutian Tang, Hareton Leung
Department of Computing
The Hong Kong Polytechnic University, Hong Kong
{csytang,cshleung}@comp.polyu.edu.hk

***Abstract*—Software Product Line Engineering is a key approach to construct applications with systematical reuse of architecture, documents and other relevant components. To migrate legacy software into a product line system, it is essential to identify the code segments that should be constructed as features from the source base. However, this could be an error-prone and complicated task, as it involves exploring a complex structure and extracting the relations between different components within a system. And normally, representing structural information of a program in a mathematical way should be a promising direction to investigate. We improve this situation by proposing a probability-based approach named *StiCProb* to capture source code fragments for feature concerned, which inherently provides a conditional probability to describe the closeness between two programming elements. In the case study, we conduct feature mining on several legacy systems, to compare our approach with other related approaches. As demonstrated in our experiment, our approach could support developers to locate features within legacy successfully with a better performance of 83% for precision and 41% for recall.**

***Index Terms*—Software product line, variability, feature mining, program slicing.**

I. Introduction

Software Product Line Engineering(SPLE) [1] is regarded as an efficient approach to provide a set of systems with tailored-made services within a domain. Successful adoption of product lines allows developers to provide applications with strong advantages in terms of time to market, maintenance efforts and development costs. With systematical reuse of code and design, a product line could generate several product variants under different configuration context. For example, in Linux kernel system, 32-bit and 64-bit processing schemas are provided to users. Simply, it can be regarded as two systems, one for 32-bit and another for 64-bit.

In product line engineering, *features* are used to describe all behaviors of a system [2]. For instance, a business transaction system is normally customized to realize different banking services in various currencies, with each service deemed as a feature. Unfortunately, given the complexity and high workload of developing many variants of a system and extra maintenance and configuration work required, this normal process is often impractical and error-prone. A more practical and simplified approach could be migrating legacy source into a product line, since most modules in legacy could be reused and limited development work is required. To construct a product line system based on a legacy source, retrieving the features and associated code would be an essential first step.

Currently, most works in constructing a product line are primarily concentrated on solutions on analyzing product lines and building product lines from an abstract aspect, for instance, from architecture level or module, including model checking, refactoring and so forth, to analyze variability in the product line [3]–[5]. The main problem of these coarse-granularity approaches in terms of constructing a product line is that they cannot recover a feature's implementation in a fine granularity matter. Like, inside a method, statements might belong to various features.

Our Contribution. In this paper, we propose a fine-granularity approach, which describes the closeness between programing elements (like, AST node) using conditional probability and the probability values are then used as indicators to guide the mining of features in our context. To further assess our approach, we developed an Eclipse plug-in tool named *Loong*[1] to obtain code fragments from the code base for the feature concerned, and we compared the performance with three other feature mining approaches, including type check, topology analysis and text comparison, with several case studies.

Organization. This paper is organized as follows. Section II provides a bird's eye view of feature mining process. Section III introduces the underlying model, and two research questions are raised. Our approach is introduced and explained in Section IV. We conduct case studies and exhibit our experimental results in Section V and VI respectively and discuss the results in Section VII. Related work is introduced in Section VIII, and we conclude our paper in Section IX.

II. Feature Mining Process Overview

The procedure of detecting potential variants from legacy could be deemed as identifying assets from an application. Particularly, in this work, we focus on deriving features' implementation from the source base.

As illustrated in **Fig.**1, the whole feature mining process consists of four steps as follows:

[1]Loong: available at http://www.chrisyttang.org/loong/

978-1-5090-5501-2/17 © 2017 IEEE

SANER 2017, Klagenfurt, Austria
Main Research

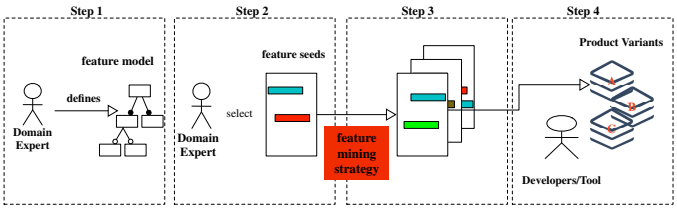


Fig. 1. Feature mining process overview

1) A domain expert models the product and describes features and their underlying relationships and constraints in a feature model.
2) Moving on, for each feature, developers have to select an initial seed to represent this feature. For example, a method named "Lock" could be used to represent feature *locking*.
3) For each feature, the feature mining strategy expands the known code range for the feature iteratively until some stopping criteria are met.
4) Finally, developers *rewrite* code fragments with different configurations. That is, selecting some variants from the variants set to generate members in the product family.

Within this process, we focus on **Step 3**, which is obtaining all code that implement a feature starting with an input seed, and a feature relation model defined by the domain expert. As mentioned in **Step 2**, with the selection of seed for each feature, the feature mining task has been transformed to finding all related code fragments iteratively based on the given seed.

III. Underlying Model

A. Basis

Programming Elements. To retrieve code fragments that describe variants and their internal relationships, we use a graph-base representation of the system, in which nodes denote programming elements and links stand for dependences. Currently most source-based tools (such as Suade [6] and Cerberus [7]) merely focus on methods and fields, which may lead to inaccurate results. In our approach, programming elements include local variables, fields, statements, types, methods, classes and interfaces. We denote the set of programming elements in a system as E. Technically, in this paper, we use abstract syntax tree(AST) nodes to represent programming elements.

Among these programming elements, relationship $(R \subseteq E \times E)$ indicates how they are linked and impact each other. *Contain* relation shows the hierarchical structure between elements. For instance, import a package or API in a class (`import java.util.Map;`). This relation could be discovered in class import (API import is covered), class instance declaration, enumeration, and inner class. *Reference* denotes a use relation, which could be method invocation, field use and type reference. For example, assigning a local variable `cfg` to a field `controlFlowGraph` using `this.controlFlowGraph = cfg`, where a field named `controlFlowGraph` is accessed and updated with local variable `cfg`. In addition, *usage* provides an indirect reference between elements, namely, one element might reference another's attributes or functions. This relationship mainly includes `cast`, `instanceof`, super and child class.

Feature. In our product-line setting, we require additional domain knowledge by defining the feature model [1], which describes how features enclosed in products are organized and their underlying dependencies and constrains. The feature model consists of a set of features (F) and relations between these features. Two fundamental relations: *mutual exclusion* and *implications* are frequently used in feature models. Specifically, *mutual exclusion* $(M \subseteq F \times F)$ denotes two features are mutually excluded and code segments belonging to one feature cannot be part of another. Whereas, *implications* $(\Rightarrow \subseteq F \times F)$, which initially come from the "if feature f is included in some variants, f's implied feature g must be covered in these variants", is useful in terms of setting seeds, since it would be redundant to provide seeds for an implied feature. Implication relation is a typical relation between features in a hierarchical relationship.

Annotation. Annotation describes the mapping between a feature and programming elements. That is, annotation $(A \subseteq E \times F)$ shows programming elements have been assigned to features during the seed selection. Each programming element could be attached to multiple features during the mining. For the relations ($\Rightarrow$ and M) mentioned, we extend the annotation as $A^* = \{(e, f) \,|\, (e, g) \in A, g \Rightarrow^* f\}$ to represent the closure of A with *implication* relation, where $\Rightarrow^*$ is the reflexive closure representation of $\Rightarrow$. In detail, a feature f's $\Rightarrow^*$ relation contains all features that implies f, that is $(g \Rightarrow^* f)$, along with f itself. Thereby, A^* relation contains two part: (1) all elements that are directly annotated to feature f as (e, f); and (2) all elements that are indirectly annotated to feature f using *implications* relation as $\{(e, f) \,|\, (e, g) \in A, g \Rightarrow f\}$.

B. Modelling Closeness between Element and Feature

RQ1: *How to measure the probability that a programming element should be annotated to a certain feature?*

Considering the whole process of feature mining approach as steps of iteratively annotating programming elements to feature, this section starts from raising a question on how to measure the probability that a programming element should be annotated to a certain feature. Moreover, we introduce the concept of annotation state and feature-element correlation coefficient for modeling this question.

Definition 1 (Annotation State)**.** An annotation state of a feature f is defined as a set of elements, that have been annotated to f is represented by

$$S\,(A^*, f, i) = \{e |\, (e, f) \in A^*\}$$

Here, i represent a certain annotation iteration. Specifically, the feature mining process for a single feature could be deemed as a series of transformation of annotation states as shown in **Fig.**3. In detail, at the beginning, seeds are selected and

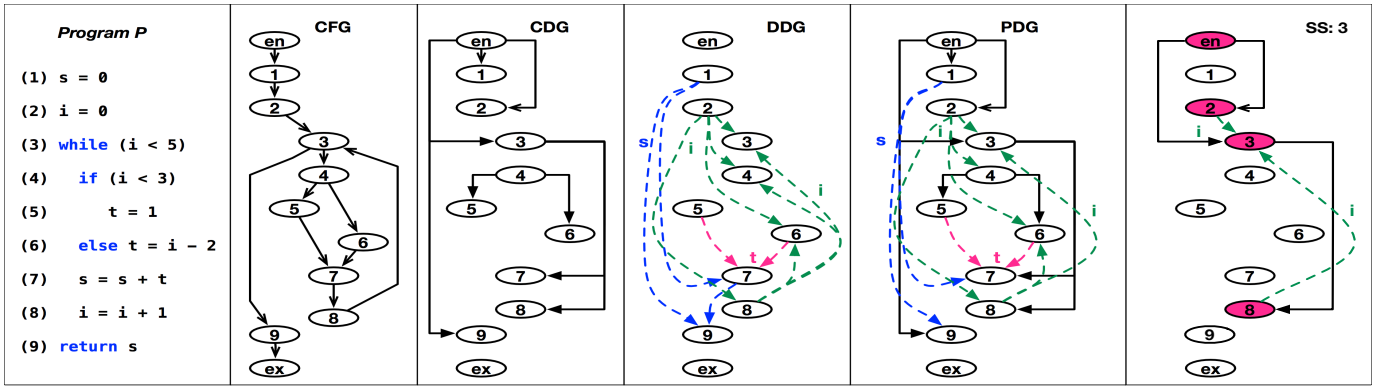


Fig. 2. Example program P with its control dependency graph, data dependency graph, program dependency graph, and the slicing scope for node 3.

annotated to the feature. Then, one or more programming elements are annotated to this feature at each iteration automatically, which transforms the current annotation state to its immediate successor, such as from $S(A^*, f, i)$ to $S(A^*, f, j)$ in the example. Thereby, the mining process for a feature f could be regarded as a series of annotation state transformation from $S(A^*, f, 0)$ to $S(A^*, f, Stp)$. The symbol $S(A^*, f, 0)$ represents the initial state in which only seeds are annotated to f. The state $S(A^*, f, Stp)$ is the final state, and it contains all code fragments that have been annotated to f when the mining process stops. For an adjacent transformation, like from annotation state i to j, the feature mining approach will look up all candidate elements, which could be annotated to the current feature and annotate those with high likelihood. Therefore, we define the feature-element correlation coefficient to express this likelihood.

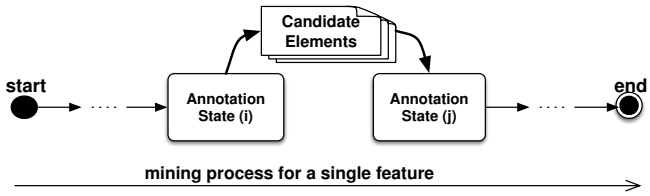


Fig. 3. The general process of feature annotation

Definition 2 (Feature-Element Correlation Coefficient)**.** A measure of the probability that a programming element e should be annotated to feature f for an annotation state $S(A^*, f, i)$, and is represented in the form of a conditional probability as

$$p(e|S(A^*, f, i))$$

Given the correlation coefficient represented as $p(e|S(A^*, f, i))$, where $S(A^*, f, i)$ is a set of programming elements at ith iteration has been annotated to f, there should be a method to measure "closeness" between two programming elements. Here, "closeness" indicates the degree that two programming elements belong to the same feature. And this conditional probability representation is designed to stimulate this "closeness".

Therefore, the feature-element correlation coefficient could be regarded as the probability required in **RQ1**. To compute this correlation coefficient, another research question, that is how to capture the "closeness" between two programming elements, should be answered first.

C. Modelling Closeness between Elements

RQ2: *How to provide a mathematical representation to capture the "closeness" between two programming elements?*

For the research question (**RQ2**) and requirements presented above, we introduce two key concepts, slicing scope and binding.

Definition 3 (Slicing Scope)**.** For a programming element, its slicing scope is defined as

$$sscope(e) = e \cup \left\{ s | s \xrightarrow{df} e \vee s \xrightarrow{cf} e, s \in E \right\},$$

where $s \xrightarrow{df} e$ represents a data dependency flow from s to e and $s \xrightarrow{cf} e$ shows a control dependency flow from s to e.

Example. In the definition of slicing scope, the control dependency graph (CDG) is a data structure which describes the control dependencies for operations in a program [8]. In addition, the data dependency graph (DDG) shows data flow dependencies between statements [9]. A program dependency graph (PDG) contains all nodes defined in CDG and DDG, and edges in PDG are all inherited from CDG and DDG. As shown in **Fig.**2, the PDG is used to compute the slicing scope. For instance, for the programming element `i<5` (line: 3), we obtain the slicing scope as a set of colored nodes $sscope(3) = \{en, 2, 3, 8\}$. The entry en is covered by referencing the control dependency and node 2 and 8 are included due to data dependency.

Theoretically, for a given a program p, the slicing scope of a programming element e in p returns a program slice with respect to a slicing criterion on that element e. Program slicing [10], [11] is a well-adopted program transformation approach with respect to a given slicing criterion. Since the definition

[1]Tool JayFX (http://cs.mcgill.ca/~swevo/jayfx/) is integrated in our tool to extract and build CDG and PDG

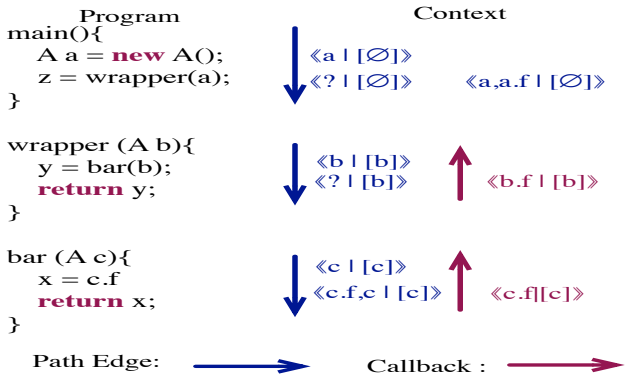


Fig. 4. Example of context transformation in method invocation

of slicing scope follows the same principle in building a program slice, program slicing serves as a theoretic footstone of concepts we defined and their extensions. Besides, it also indicates that approaches for building the program slice could be reused to obtain the slicing scope in our paper. Specifically, we use the program slicing approach defined in [10] to find the slicing scope with a given programming element.

Definition 4 (Binding)**.** For a programming element e, its binding $bind\,(e)$ is defined as all variables and fields defined or used in e's slicing scope as

$$bind(e) = def(e) \cup use(e)$$

Intuitively, our binding definition of a programming element e could represent e in a broad sense, considering all variables or fields, for both use and define, are covered. And their types are covered, since the type information is binding with the programming element. However, it is still insufficient to describe a complex relation between two programming elements (m, n) by just using their bindings $(bind\,(m)\,, bind\,(n))$. Such as, method invocation, class inheritance, and method overriding. To resolve this, we further reinforce the binding definition by adding another factor "input context" to describe how given one programming element affects the current element. For example, for a call from method m to n, the call site in m is an "input context" for n. And other methods may also invoke n, and each caller brings its unique "input context" to n.

Definition 5 (Context Binding)**.** For a programming element e, its context binding $contbind(e, [c])$ is defined as all variables and fields defined $(def\,(e))$ or used $(use\,(e))$ in e's slicing scope under an input context from programming element c.

In context binding, *context* gives an unique identifier for a programming element in terms of runtime. For example, given two context $a.f()$ and $b.f()$ give two different cases: one is function f is invoked by instance a and another is by b. Note that the inherent properties are dependent on object-oriented languages, as different programming languages have their own unique specification for implementing them. For example, in Python, multiple inheritance from classes is allowed, which is not allowed in Java. Here, we use the language specification defined in Java for illustration.

Method Invocation. Method invocation could bring context change especially when parameters are passed [12] from a call site to its callee. A callsite `l=`r_0`.m(`$r_1, ..., r_n$`)` will connect the parameters in the call site with arguments in invoked method `m` and `m` receives the *run-time* parameters $(r_1, ..., r_n)$ from this call site. Therefore, the invoked method obtains a unique input context from the call site. To resolve this, we follow Andersen's [13] context-aware analysis and dispatch the binding of the call site into callee as:

$$contbind(m, [r_1, ..., r_n]) = dispatch\,(p_i = r_i) \rightarrow bind\,(m)\,,$$

where the context $[r_1, r_2, ...r_n]$ is dispatched to method m by mapping all parameters in the call site to arguments in the callee.

Example. As the example shown in **Fig.**4, we use the label $(\ll curbind|\,[context]\,\gg)$ to mark binding computed at a program point. In this tag, $curbind$ represents the context binding collected at this program point, and $[context]$ shows the input context to this method. For example, in **Fig.**4, method `main` gives a context $[a]$ to its callee `wrapper`. In `wrapper`, this context $[a]$ is dispatched to the method body of `wrapper` as $a \rightarrow b$. And this initial context $[a]$ will continue passing to `bar` from the method `wrapper`. Take the method `bar` as an example, its context binding $contbind\,(bar, [a])$, which represents the call path `main`→`wrapper`→`bar`, is $\{a.f, a\}$. Here we use $[a]$ just to provide a simplified representation of the context given by `main`; in practice, we use unique identifiers to encode contexts from different sources.

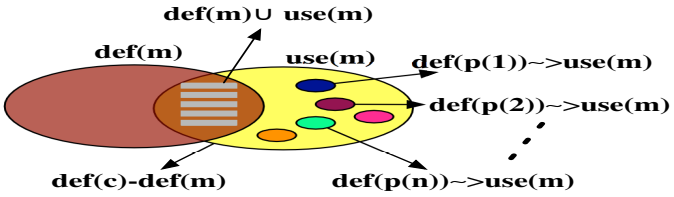


Fig. 5. Example of context binding in overriding

Overriding. The context of an overriding method m defined in class c, is defined by all classes or interfaces that c inherited from. The context binding of a method overriding $contbind\,(m)$ in a class c contains two parts: (1) programming elements defined and used in m, and (2) elements defined in c's parent class or interface and used in m. Therefore, a general representation of the context binding of an overriding method could be denoted as:

$$contbind\,(m, [p_1, \cdots, p_n]) = def\,(m) \cup \bigcup_{i=1}^{n} (use(m) \rightsquigarrow def\,(p_i))$$

We illustrate this formula using the example in **Fig.**5. Based on our definition on binding, it contains two parts: the variables and fields defined and used in m. Particularly, the variables defined in the overriding method could be directly represented

as $def\,(m)$. Whereas, for the variables and fields used in m, they potentially come from three sources: (1) variables defined and used in method m as shown in the overlap area; (2) fields defined in class c as shown by $def\,(c) - def\,(m)$; and (3) all fields inherited from its parent classes and interfaces $p_1, p_2, \cdots p_n$, where the symbol $\rightsquigarrow$ is used to specify the source of the context. For example, $def\,(p\,(i)) \rightsquigarrow def\,(m)$ indicates all variables or fields used in m but defined in p_i.

Example. Considering a fragment of overriding given in **Fig.**6, class `FlyingCar` is inherited from interface `OperateCar`. `startEngine`'s binding contains two "encryptedValue" in different contexts, one is defined in `FlyingCar.startEngine`, and another in `OperateCar`. Thereby, the context binding of method `startEngine` ($contextbind$(`startEngine`)) should be {`OperateCar.encryptedValue`,`encryptedValue`}.

```
1 public class FlyingCar implements OperateCar {
2   public int startEngine(int encryptedValue) {
3     OperateCar.super.startEngine(OperateCar.encryptedValue);
4   }
5 }
-------------------------------------------------
6 public interface OperateCar {
7   int encryptedValue = 1;
8   default public int startEngine(int value) {...}
9 }
```

Fig. 6. A sample code of overriding

Inheritance. Different from overriding, in inheritance, we are interested in providing a context binding for the inherited class. The context binding of the inherited class c consists of the binding in c and all fields defined in all its parent classes and interfaces that c inherited from. It is defined as:

$$contbind\,(c, [p_1, \cdots, p_n]) = bind\,(c) \cup \bigcup_{i=1}^{n} def(p_i),$$

where $def(p_i)$ represents all fields defined in p_i. The context binding of inherited class contains all fields inherited from p_i, along with all the variables and fields originally defined in c, as $bind\,(c)$.

Now, we are able to provide a representation of an annotation status $S\,(A^*, f, i)$ of the feature f as

$$S\,(A^*, f, i) = \bigcup_{a}^{a \in S(A^*, f, i)} contextbind(a),$$

which is a collection of context bindings of all programming elements within the annotation status $S\,(A^*, f, i)$. However, in our model, $p\,(e|S\,(A^*, f, i))$ is an exact value to indicate the probability that a programming element belongs to the feature f based on the annotation status $S\,(A^*, f, i)$. Therefore, in the coming section, we will show how our *StiCProb* works in exploring code fragments for features, and how the condition probability $p\,(e|S\,(A^*, f, i))$ serves as a major component of the mining process.

IV. *StiCProb* Approach

We first provide an overview of the whole *StiCProb* approach, and then all its steps. Specifically, it contains three steps.

1) The first step is to build a database for the system, which contains all programming elements and underlying relations between them. This part has been covered in Section III-A;
2) The second step is to build a uniqueness table, in which the major task is to show the uniqueness between two programming elements with a relation.
3) At last, we use the feature-element correlation coefficient defined as an indicator to mine code fragments for features concerned.

The key characteristics of our approach is it learns the probability from the context of each programming element (step 2) to seek potential elements to annotate. For example, given a call relation from method m to n and another call from j to n, if there are 5 call relations start from m and only 2 call relations start from j, j should be more unique to n in terms of call relation. And *StiCProb* is able to collect this kind of context information, which further contributes to the feature mining.

Knowing that **RQ1** can be answered only if **RQ2** is answered, this section starts from solving **RQ2**.

A. Building a Uniqueness Table.

We reserve a uniqueness table for the system to describe the cross uniqueness for any two programming element(s,t) under a relation r ($s \xrightarrow{r} t$). In detail, a relation table could be represented as a five-tuple $U\,(E, T, R, P_{forward}, P_{backward})$. For a specific element $u \in U$, it is represented as $u\,(s, t, r, p_{forward}, p_{backward})$, which means there is a relation r from the programming element s to t. The uniqueness of $\underline{t \text{ to } s}$ for relation r if s has been annotated to feature f is represented by $p_{forward}$. Thereby, we define probability $p_{forward}$ as

$$p_{forward}\left(s \xrightarrow{r} t | (s, f) \in A^*\right) = \frac{contbind\,(t, [s])}{contbind\,(s)},$$

where $contbind\,(t, [s])$ represents the context binding of t given a context from s according to our previous definition on $contbind$. Therefore, the definition of $p_{forward}$ could show the uniqueness of $\underline{s \text{ to } t}$ for relation r. We could define the probability $p_{backward}$ as

$$p_{backward}\left(s \xrightarrow{r} t | (t, f) \in A^*\right) = \frac{contbind\,(t, [s])}{\bigcup_{i \xrightarrow{r} t} contbind(t, [i])},$$

where $\bigcup_{i \xrightarrow{r} t} contbind(t, [i])$ represents a collection of context binding from all programming elements, which have relation r with t.

Example. As shown in **Fig.**7, for the call from s to t ($s \xrightarrow{call} t$), the forward probability $p_{forward}\left(s \xrightarrow{call} t | (s, f) \in A^*\right)$ describes the uniqueness of t to s. That is, there might be multiple call relations starting from s as shown in the example, the

value of $p_{forward}$ indicates the weight of the call from s to t in terms of all method call relations starting from s. On contrast, the backward probability $p_{backward}\left(s \xrightarrow{call} t|(t,f) \in A^*\right)$ depicts the uniqueness of s to t as the weight of the call from s to t referencing all method-call relations that end with t.

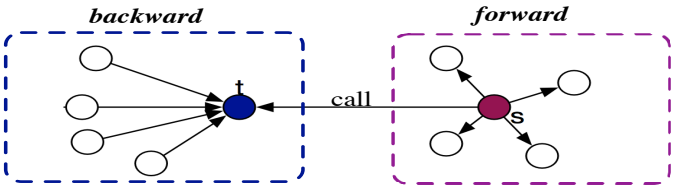


Fig. 7. An example of a call relation

The strength of forward and backward probability for a relation ($s \xrightarrow{r} t$) from s to t is that they can capture s and r's surrounding information respectively. Specifically, the backward probability $p_{backward}$ could explore the relative context information of t as the left side shown in **Fig.**7. And the forward probability $p_{forward}$ could explore the relative context information of s as the right side shown in **Fig.**7. Thereby, with the forward and backward probability, the research question **RQ2** is solved. Moving on, we will answer **RQ1** by introducing the detail of *StiCProb*.

B. StiCProb

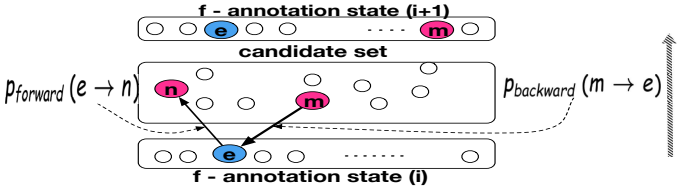


Fig. 8. Illustration of *StiCProb*

We first use **Fig.**8 to illustrate the underlying idea of *StiCProb*. As introduced previously (see Section III), the mining process for a feature could be regarded as a series of transformation for annotation state. For a feature f, at the beginning, seeds are selected for this feature, which gives the initial annotation state of f as $S(A^*, f) = seeds$. Iteratively, the annotation state transforms from one to another and at each transformation one or more programming elements are annotated to f. For an annotate state transformation, as shown in **Fig.**8, the candidate set covers all elements that have direct relations with elements in annotation state (i). And then *StiCProb* categorizes the candidate set into two parts. The first part is relation starts from an element in the candidate set, like $m \rightarrow e$, with the backward probability $p_{backward}$ to represent the closeness of this relation, which indicates the uniqueness of m to e. Another is relation starts from elements in the annotation state and ends with an element inside the candidate set, such as $e \rightarrow n$, with the forward probability $p_{forward}$ to describe this relation, which shows the uniqueness of n to e. In other words, *StiCProb* assesses all candidates by giving a probability description of how they are unique to elements in the current annotation state.

The pseudo code description of *StiCProb* approach is shown in **Alg.**1. The detailed introduction of *StiCProb* is separated into three components: input, main procedure and output.

Algorithm 1: StiCProb feature mining approach

Input: $seeds$, fm, $threshold$, U
Output: all annotation states for features $Sset$ in fm

```
1  Create a set of annotation states as
   Sset = ⋃_f^{f∈features} S(A*, f);
2  Assign seeds to each feature as S(A*, f) = seeds(f);
3  Create feature set features with all features in fm;
4  while features not NULL do
5      for feature f in features do
6          Create set waitList = ∅;
7          Create candidate set C(S, f) = ∅ for f;
8          Add all elements have relations with elements in
           S(A*, f) to C(S, f) ;        // initialize C(S, f)
9          for element m in C(S, f) do
10             if there is a relation r from m to the element
               e in S(A*, f) then
11                 Let value =
                   p_backward(m -r-> e|(e, f) ∈ A*);
12             else
13                 Let value =
                   p_forward(e -r-> m|(e, f) ∈ A*);
14             if value > threshold then
15                 Add m to waitList;
16         Update S(A*, f) ← S(A*, f) ∪ waitList;
17         if StopCheck(f) is TRUE then
18             Remove f from features;
19 return Sset;
```

Input. The input for algorithm *StiCProb* is the program, containing:

1) Feature model (fm): A feature model is given to show all features required to mine and their underlying relations. Proposing approaches to obtain the feature model is out of scope of this paper. In the case study, we adopt the feature model defined for subject systems in other research works;
2) Feature seeds ($seeds$): The seeds (represented by $seeds$ in algorithm) selected for each feature. For each feature, one or more programming elements are selected as seeds to represent a feature. We leave the discussion on how to select seeds for a feature in section V-D: *Experimental Setting*;
3) $threshold$: It is used to decide whether a programming element could be annotated to a feature;

4) Uniqueness table (U): The uniqueness table U is built for all element-relation tuples (m, n, r), where there is a relation r from m to n.

Main procedure. We will introduce how *StiCProb* contributes to feature mining. Due to the length of the algorithm, we separate it into four sections and present them respectively.

1) Line $1-3$: For each feature, an annotation state $S(A^*, f, i)$ is created. And a set $Sset$ is used to store all these annotation states. Each annotation state is initialized with seeds for feature f.
2) Line $4-8$: For each feature, a candidate set C is created by adding all elements having relations with the elements in the current annotation state S. And relations could be covered in both directions. That is, an element that either has a relation target at an element inside S or has a relation from an element inside S should be covered in C.
3) Line $9-15$: Following the previous step, it iterates over all elements in the candidate set. If there is a relation from an element m within the candidate set to an element e in the annotation state (Line $10-11$), the backward probability $p_{backward}$ is used to capture the relation. For opposite direction, the forward probability is used. In addition, Line $10-13$ is the kernel of *StiCProb*, since it shows how the feature-element correlation coefficient (see **Def.**2) is implemented in our approach. It also gives the answer to research question **RQ1**.
4) Line $16-18$: Line 16 will update the annotation state for each feature by adding the $waitList$, which contains all elements that should be annotated to this feature. The rest (Line $17-18$) checks whether it can stop the current mining process using a function $StopCheck(f)$. The concrete description of function $StopCheck$ is introduced in section IV-C: *Stopping Criteria*.

Output. The output returns the set ($Sset$) of all annotation states for all features in the feature model. At this step, the feature mining process for all features are finished and each annotation state $S(A^*, f, Stp)$ gives all elements that have been annotated to feature f.

C. Stopping Criteria

Stopping criteria is shown as the $StopCheck$ function in **Alg.**1. In this paper, we use the $threshold$ as an indicator to stop the mining process for a feature. For a feature f, if all $value$ (Line $11, 13$) computed in algorithm are lower than the threshold defined, the mining process is halted. The value $value$ is computed based on either forward or backward probability to determine whether a candidate element should be annotated to a certain feature.

V. Case Studies

A. Subject Systems

Note that not all legacy systems are qualified for our experiment, as we need a specific benchmark for the system be available. The benchmark contains a set of files that describe how programming elements are mapped to features. Without the benchmark, it would not be possible to assess the performance of our approach. Thereby, we use those systems that have been analyzed and learned in other works to exclude the bias in creating the benchmark on our own. However, this, in return, limits the scope of subject systems. As a result, we carefully select four different subject systems that have been developed and well-researched by others, from academic and industrial systems.

- **Prevayler**[2]. An open-source object persistence library for Java with 8009 LOC. It is a well recognized product for product line research [14], [15], although it is not originally developed as a product line application. It contains five features: *Censor*, *Gzip*,*Monitor*, *Replication*, and *Snapshot* with a dependency $Censor \Rightarrow Snapshot$.
- **MobileMedia**. Originally developed by University of Lancester, UK as a product line with 4653 LOC [16]. It contains several features: *Photo, Music, SMS Transfer, Copy Media, Favourites*, and *Sorting*. The dependencies include: $Photo \vee Music$, $SMSTransfer \Rightarrow Photo$ and $MediaTransfer \Leftrightarrow (SMS\,Transfer \vee Copy\,Media)$.
- **Lampiro**[3]. An open-source instant-messaging client with 44584 LOC. Here feature *Compression* without dependency is selected, as others are affected by limited code fragments or cannot be deemed as debugging features. Here, debugging feature represents feature provides a "invokable" service to end-users rather than assisting the workflow. Some other small features have already been tested in other cases. Lampiro is a qualified candidate because it is originally developed as a product line with conditional compilation.
- **ArgoUML**[4] with 120 KLOC, provides modeling support for UML 1.4 diagram and supports multiple programming languages. In ArgoUML, following seven features are selected: *Cognitive*, *Activity Diagram*, *StateDiagram*, *Collaboration Diagram*, *Sequence Diagram*, *Use Case Diagram*, and *Deployment Diagram* from an empirical research [17]. The feature *Logging* is not covered in our mining work, as it is not a callable feature for end customers. In another study [14], a dependency $ActivityDiagram \Rightarrow StateDiagram$ is added. In our experiment, we adopt this setting.

B. Related Approaches for Comparison

We carefully select 3 related approaches used in [18] for performance comparison.

- **Type System.** Type system [19] is initially designed to bring a type-checking system to product line that ensures all variant products generated are type safe. It has been re-implemented to cope with the feature mining task,

[2]Prevayler: available at http://prevayler.org
[3]Lampiro v9.6.0 available at https://code.google.com/archive/p/lampiro/
[4]ArgoUML: available at: http://argouml.tigris.org

and the underlying idea is to look up definition from references. For example, if a type reference is annotated to a feature f, the type declaration of this type should also be annotated to f. The type system checks all these relations, such as, from method invocation to declaration, from variable/field access to its declaration, and from type access to its declaration.

- **Topology Analysis.** Originally designed by Robillard [6] and adjusted to the feature mining task in [18]. Topology analysis explores all structural neighbors, such as caller method and related fields, for a given programming element. It then ranks related programming elements according to two metrics *specificity* and *reinforcement*.
- **Text Comparison.** Text comparison defined in [18] reserves a *vocabulary* list for each feature. It ranks the substring based on a relative weight and its occurrences.

C. Implementation

We have implemented our *StiCProb* and other related approaches in an Eclipse plug-in tool named *Loong* following the feature mining process defined in **Fig.**1. We have released the source code, a full tutorial for this tool, and experimental data on the project host page: http://www.chrisyttang.org/loong/.

D. Experimental Setting

1) Defining Feature Model and Selecting Seeds: In principle, a domain expert should be involved in the experiment and contribute to two parts: (1) defining the feature model for the legacy system. That is, exploring features existed in the system and the way they are organized and influence each other; and (2) helping to select seeds for each feature. Therefore, the domain knowledge on features and their relation will have a significant impact on the performance of feature mining approach. Considering our target is to verify *StiCProb*'s performance, it will be wise to reduce all human bias by using the feature model that has been well adapted and learned by other research works and selecting seeds using automatic tools. In this paper, we use the feature model built by other research works and a tool named FLAT^3 [20] to obtain the seeds for each feature. This is done to exclude bias, and make the semi-automatic (see Section III) process repeatable.

2) Other Settings: We list settings for other factors which could influence the performance of feature mining approach as follows.

- **Number of seeds for each feature.** As mentioned we use *FLAT*^{3} to provide seeds for each feature. In the experiment, we select the top three items returned by *FLAT*^{3}.
- **Threshold.** In *StiCProb*, we use a threshold of 0.6. Intuitively, a higher threshold will make the annotation more precise and a lower one will annotate more programming elements. Here, in the experiment, the threshold is set to a median value and its influence on performance is discussed later in section VII.

The first setting (number of seeds) is suitable for all approaches, including our *StiCProb* and other three related approaches. However, the second setting for threshold is specific to *StiCProb*. With all these settings, we try to exclude the possible biases in conducting our experiment in selecting seeds and subject systems.

VI. Experimental Result

A. Measurement

Our definitions for recall and precision are specific to the feature mining procedure. Our framework returns all lines that could belong to the features at the statement level. In a *binary* mapping context, a single statement could either belong to a feature or not. Therefore, the recall should measure how features' implementations are recovered. On contrast, the precision measures the degree that a feature mining algorithm could provide correct recommendations. In our framework, precision and recall are defined as:

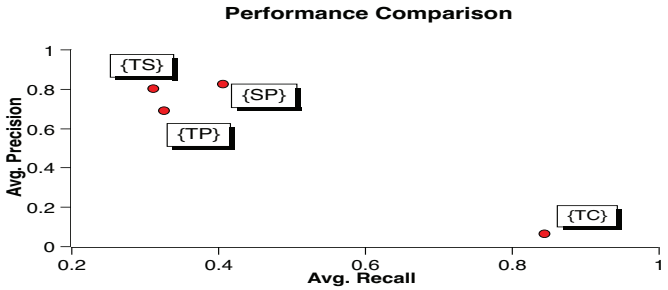
$$Precision = \frac{Correct\ recommendations}{All\ recommendations\ provided} \quad (1)$$

$$Recall = \frac{Correct\ recommendations}{Lines\ of\ code\ annotated\ in\ benchmark} \quad (2)$$

Another indicator for comparison is *f-measure*, which measures the performance of a model regarding to both precision and recall as: f-measure = $\frac{2*Precision*Recall}{Precision+Recall}$

B. Result

The experimental result is shown in **Tab.**I, where *StiCProb* receives an average precision of 83% and an average recall of 41% on four subject systems with a threshold of 0.6. Furthermore, we compare *SticProb* with other approaches on all four systems with the same experimental settings as depicted in **Fig.**9. This performance generally gives a better result comparing to type system (pre.:80%, recall:22%), topology (pre.:69%, recall:33%), and text comparison (pre. 6%, recall: 84%) average.



TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

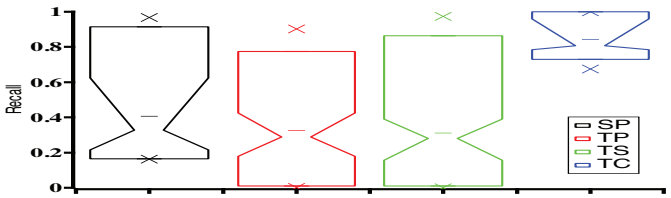
Fig. 9. Performance Comparison on Subject Systems

For the *f-measure* shown in **Tab.**II, *StiCProb* returns a competitive performance in terms of both precision and recall.

TABLE I
STICPROB PERFORMANCE WITH $threshold = 0.6$

Project	Feature	Feature Size LOC	FR	FI	Mining Results IT	Recall	Prec.	Project	Feature	Feature Size LOC	FR	FI	Mining Results IT	Recall	Prec.
Prevayler	Censor	105	10	5	3	17%	60%	MobileM.	M.Transfer	153	4	3	14	97%	94%
	Gzip	165	4	4	3	16%	100%	Lampiro	Compre.	5155	33	20	34	40%	82%
	Monitor	240	19	8	2	17%	82%	ArgoUML	Cognitive	16319	285	233	127	70%	92%
	Replication	1487	37	28	26	79%	98%		Activity	2282	115	80	17	26%	74%
	Snapshot	263	29	5	9	42%	99%		State	3917	115	88	18	33%	82%
MobileM.	CopyMedia	79	18	6	4	43%	95%		Collab.	1579	53	40	40	17%	72%
	Sorting	85	20	6	4	32%	100%		Sequence	5379	65	53	98	33%	89%
	Favorites	63	18	6	12	20%	100%		Use-Case	2712	59	49	39	19%	70%
	SMS Trans.	714	26	14	23	91%	49%		Deployment	3147	57	47	36	22%	67%
	Music	709	38	16	4	39%	90%								
	Photo	493	35	13	5	63%	61%								

LOC: lines of code, FR: count of distinct code fragments;IT: number of iteration, Prec.: precision



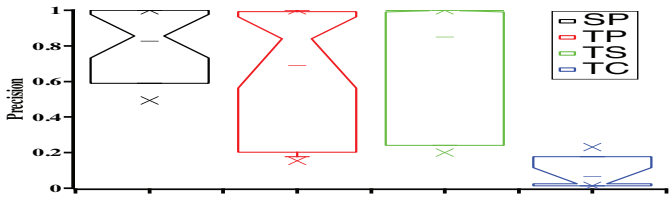
TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

Fig. 10. Method comparison using notched box plot in recall

TABLE II
$f - measure$ ON ALL APPROACHES

	SP	TS	TP	TC
$f - measure$	0.55	0.45	0.44	0.12

In addition, as the notched box plot for recall shown in **Fig.**10, at 95% confidence interval of median, *StiCProb* performs better than both type system and topology analysis for most cases. From another aspect, the notched box plot for precision in **Fig.**11 indicates that *StiCProb* works better than both topology analysis and text comparison.



TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

Fig. 11. Method comparison using notched box plot in precision

Runtime Performance. We also evaluate the performance in terms of run-time. We test all algorithms on a Mac(10.12) machine with Intel i5 2.6GHz, 8G 1600 MHz DDR3, and targeting on Eclipse 4.5 with JRE 7. The result is shown in **Tab.**III.

TABLE III
RUNTIME PERFORMANCE (SECOND)

	TS	SP	TP	TC
Pervalyer	1	2	2	71
MobileMedia	2	2	3	21
Lampiro	1	29	13	135
ArgoUML	254	1500	1980	5415

In summary, based on the recall performance, all methods could be ranked as TC (0.77,0.80,0.93) $\gg$ $\boldsymbol{SP}$ **(0.29,0.33,0.53)** $\gg$ TP (0.12,0.32,0.41) $\gg$ TS (0.12,0.21,0.37). For precision performance, TS (0.66,0.92,1) $\gg$$\boldsymbol{SP}$ **(0.71,0.85,0.96)**$\gg$ TP (0.42,0.84,0.95) $\gg$ TC (0.02,0.02,0.11). Here, we use a three-element tuple $(first, median, third)$ to indicate the first quartile($first$), the median value($median$) and the third quartile($third$) of data. We can conclude that *StiCProb* could return a competitive and stable performance comparing to others. However, sometimes, *StriProb* spends extra-time in generating binding and contexts, which could be a potential drawback.

VII. DISCUSSION

Beyond the default settings, we investigate how the two independent variables, seeds and threshold, influence the performance. Due to space restriction, we provide a general discussion on these factors and put the details on the project webpage[5].

A. Seeds

In our feature-mining process, the seeds could strongly influence the performance. In our experiment, we adopted first three items returned by FLAT^3. By increasing the number of seeds, the performance can hardly be improved and sometimes becomes significantly worse. After a careful inspection on seeds, we discovered the following principles, which could be used to guide developers in seeds selection. First, seeds recommended by FLAT^3 might not be correct, which causes the feature mining strategy performs poorly. That is, if the quality of seeds can be improved, the performance may improve. Second, the seeds in coarse granularity could improve the recall, but sometimes at the cost of precision.

[5]http://www.chrisyttang.org/loong/

B. Threshold

In *StiCProb*, we select a `threshold` of 0.6 as the stopping criteria. That is, for an iteration, if all candidates cannot reach the threshold, the mining process for the current feature will stop. Intuitively, by setting a higher threshold, the precision can reach a higher value, and the recall drops down. However, we found that increasing the threshold, the precision is not significantly improved. For example, in *Prevalyer*, with a change of threshold from 0.6 to 0.8, the precision merely increases to 85% from 83%. That is mainly due to the use of forward and backward probabilities. And it makes the *threshold* contributes less to the performance since the forward and backward probabilities are directly decided by the structure of the system.

C. Threats to Validity

Construct and Internal Validity. The measure of performance is based on the quality of benchmarks. The benchmarks are selected from systems that have been researched by others. Nevertheless, it is possible that the selected benchmarks might not be entirely accurate. The measurements on recall and precision are based on the line of code, which are intuitively reasonable.

External Validity. (1) Due to the relatively small number of cases selected and the size of subject systems (4-120KLOC), the experimental results could not be generalized to all systems. However, this is mainly because we can only select systems that have already been researched to obtain the benchmarks. In addition, the two systems (Lampiro and MobileMedia) that are initially developed as a product line system might bring bias on performance, considering their architectures could be well-organized, like following certain design patterns, and might make feature mining approach performs well. (2) For each system, as seeds are decided using *FLAT*^{3}, it excludes the bias from selecting seeds by experimenters. Moreover, the number of seeds and threshold used in our approach could affect the performance, but we have discussed the impact of each earlier. (3) To assess the performance, we use benchmarks from others' work, which eliminates the bias introduced by providing benchmarks on our own.

VIII. Related Work

We organize all related work into three groups: feature location techniques, asset mining, and feature mining tools.

Feature Location Techniques. Feature mining in the software product line context is very different from the normal feature mining process, which aims to identify the location in the source base for a functionality concerned. The work in product line context should also take the variability and all underlying dependencies and constrains into consideration. The techniques used in feature location are mainly static analysis [21], [22], dynamic analysis [23], [24] and hybrid strategies [7]. Besides relying on the program structure, some approaches treat the program from a textual aspect [20] or exploring some dependency relations [7], [21], [25]. Due to space limitation, here, we only introduce two works [26], [27] that are highly related to our work using probability ranking. A complete literature review for feature location could be found at [28].

Poshyvanyk et al.'s work [26] is essentially a combined approach of dynamic information from execution scenarios and textual information to locate features. Antoniol's work [27] combines dynamic and static data to identify the relevant methods. Different from these two approaches, which are merely suitable for method rather than all types, our approach also contributes the fine granularity elements, like statement and local variable. Another difference between *StiCProb* and these approaches is the probability in these two approaches are obtained by tracing the runtime information and our approach collects probability from a static aspect. That is, these two approaches are dynamic approaches, and our approach is a static one, which collects information from the structure of the program.

Asset Mining. Feature mining is sometimes named *asset mining* [29]–[31]. The works in asset mining are mainly regarding to recover variant relation and model by locating, documenting and investing features in the feature model. These works contribute to what to mine, and what should be considered in the procedure. They could be deemed as a preliminary work to our contribution as the **Step 1** in our process, and we replace the process by adopting existing feature models.

Feature Mining Tools. Two tools are closely related to our work: *LEADT* [18] and *CIDE+* [14]. In *LEADT* and *CIDE+*, the authors pursued the same task on finding the feature code at a fine granularity. Our work basically contains all strategies in *LEADT* and adds our own *StiCProb* approach. On the contrast, the work in *CIDE+* mainly depends on type-checking-like mechanism assist with Cerberus's dependency analysis [7]. Differently, in *CIDE+*, the feature dependency is not explored, and it requires a large number of seeds to reach an acceptable performance.

IX. Conclusion

Product line engineering has been broadly adopted to developing applications with high customization at a low cost. To reduce the barrier in adopting product lines by migrating legacy software, we provide a novel approach named *SticProb* to extract related code fragments for feature concerned with a tool named *Loong*. *SticProb* uses the conditional probability to direct the feature mining process. Unlike all other approaches, *SticProb* can learn the environment of a programming element before annotating it to a feature. In this way, *SticProb* performs competitively in both precision and recall.

In the future work, we intend to extend our approach with support by importing dynamic analysis mechanisms. Especially, it would be attractive to use test cases to detect code fragments for features along with feature interaction.

REFERENCES

[1] K. Pohl, G. Böckle, and F. v. d. Linden, "Software product line engineering foundations, principles, and techniques," 2005, includes bibliographical references (p. [445]-456) and index.

[2] K. Czarnecki and U. Eisenecker, *Generative programming : methods, tools, and applications*. Boston [Mass.: Boston Mass. : Addison Wesley, c2000., 2000, includes bibliographical references and index.

[3] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," pp. 805–824, 2011.

[4] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," pp. 211–220, 2006.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," pp. 335–344, 2010.

[6] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, 2008.

[7] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Gueheneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, Conference Proceedings, pp. 53–62.

[8] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Berlin ; New York: Springer, 1999, includes bibliographical references and index.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[10] S. Blazy, A. Maroneze, and D. Pichardie, "Verified validation of program slicing," pp. 109–117, 2015.

[11] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–41, 2012.

[12] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, pp. 1–41, 2005.

[13] L. O. Andersen, "Program analysis and specialization for the c programming language," Thesis, 1994.

[14] M. T. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 737–754, July 2012.

[15] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," Shanghai, China, pp. 112–121, 2006.

[16] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas, "Evolving software product lines with aspects: An empirical study on design stability," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 261–270.

[17] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, Conference Proceedings, pp. 191–200.

[18] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 67–82, 2014.

[19] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 3, pp. 1–39, 2012.

[20] T. Savage, M. Revelle, and D. Poshyvanyk, "Flat^3: feature location and textual tracing tool," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 255–258.

[21] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," pp. 482–498, 1993.

[22] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," pp. 10–19, 2009.

[23] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[24] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.

[25] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," pp. 111–120, 2011.

[26] 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 Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.

[27] G. Antoniol and Y. G. Gueheneuc, "Feature identification: An epidemiological metaphor," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 627–641, Sept 2006.

[28] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[29] J. Bayer, Jean-Fran, o. Girard, M. W, rthner, J.-M. DeBaud, and M. Apel, "Transitioning legacy assets to a product line architecture," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 446–463, 1999.

[30] J. Bergey, L. Brien, and D. Smith, "Mining existing assets for software product lines," 2000.

[31] D. Smith, L. Brien, and J. Bergey, "Mining components for a software architecture and a product line: the options analysis for reengineering (oar) method," p. 728, 2001.