Constructing Feature Model by Identifying Variability-aware Modules

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

***Abstract*—Modeling variability, known as building feature models, should be an essential step in the whole process of product line development, maintenance and testing. The work on feature model recovery serves as a foundation and further contributes to product line development and variability-aware analysis. Different from the architecture recovery process even though they somewhat share the same process, the variability is not considered in all architecture recovery techniques. In this paper, we proposed a feature model recovery technique *VMS*, which gives a variability-aware analysis on the program and further constructs modules for feature model mining. With our work, we bring the variability information into architecture and build the feature model directly from the source base. Our experimental results suggest that our approach performs competitively and outperforms six other representative approaches for architecture recovery.**

***Index Terms*—feature model recovery; variability-aware modularity; feature modules; configuration; product line;**

I. Introduction

Software Product Line (SPL) is regarded as an efficient means for constructing software products within the same domain that contains multiple customized assets [1]. Successful adoptions of product lines could assist stakeholders provide applications with low costs, fast time to market and high quality, since code and designs are highly reused in each variant within a product family [1], [2].

Variants in a product family are distinguished from *features*; the domain experts are required to analyze the domain and describe the features needed and set the potential relations and constraints between these features. For instance, feature *receive*, *send* and different *email classification* strategies should be explored for an email product line. Despite the product line has been broadly adopted as a paradigm in product developing, the complexity of work is still a major risk to undertake for companies. Furthermore, even if companies start from a legacy system, there are yet many challenges, including recovering a feature model, mapping feature to its implementation, and refactoring code fragments into product variants [3].

Among all challenges, the fundamental step to resolve is to construct a feature model for the legacy system. A feature model, in a product line, represents a digram contains all features along with underlying dependencies and constraints [1]. Explicitly, the feature model can be recovered either from requirement specification [4] or code base. Thereby, recovering feature model from legacy code is a primary step to construct a product line system from a legacy. This task is paramount for working on an open-source project considering the requirement specification is normally unreachable in that situation. Unfortunately, current work on recovering feature model from source is mainly focused on recovering the feature model from a collection of product variants instead of legacy [5]. To cope with this issue, we aim at providing a semi-automatic approach to explore legacy source code and construct a feature model for the system, as a fully automatic approach is unrealistic since the user has to determine the features needed at least. Specifically, in this work, we target on Java, and it can be extended to most object-oriented languages. However, it is designed for `C` and `CPP`, which use the preprocessor to implement the variability.

Motivation. Constructing feature model is highly related to research work on architecture recovery [6], [7], program understanding [8]–[10], feature identification [11] and other relative subfields. However, it is especially different from other in following aspects: (1) it recovers the architecture in a variability fashion, that means the variability should also be mined during the process; (2) apart from the hierarchical relations between features, dependencies and constraints should also be discovered, which may affect those features without any hierarchical relation; and (3) in architecture recovery, both functional and non-functional requirements are concerned, but in a product line only functional requirements are considered.

Our Contributions. The main contributions of this paper are listed as follows:

- We define a variability-aware module system; moreover, we give a set of definition and a series of constraints to check whether a module is well-formed;
- We give a variational representation of the program to further define a set of similarity measurement to assist in merging modules into features; and
- We provide a comprehensive comparison with six representative approaches for architecture recovery by investigating four systems and compare performance from four aspects by constructing over **80** experiments!

Organization. This paper is organized as follows. Section II shows our variability-aware module system. Furthermore, Section III defines a variability-aware program dependency graph to further support our module system; our VMS approach is introduced in Section IV. Finally, Section V and VI exhibit

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.21

TABLE I
NOTATION OF MODULE MODELING

Notation	Remark
$e \in \mathcal{E}$	expressions
$t \in \mathcal{T}$	types
$x \in \mathcal{X}$	function names
$o \in \mathcal{O}$	configuration options
$c \in \mathcal{C} = 2^{\mathcal{O}}$	configurations
$v \in \mathcal{V} = 2^{\mathcal{C}}$	variability model
$\Gamma \in \mathcal{C} \xrightarrow{p} \mathcal{X} \xrightarrow{p} \mathcal{T}$	import function signature
$\Delta \in \mathcal{C} \xrightarrow{p} \mathcal{X} \xrightarrow{p} \mathcal{T} \times \mathcal{E}$	function defintion
$m = (v, i, j, \Gamma, \Delta) \in \mathcal{M}; i, j \subseteq \mathcal{O}$	module[1]
$\phi_{DUC \wedge TC}\,(m)$	constraint function

case study and experimental results. We discuss the results in Section VII, and related work in Section VIII. We conclude this paper in Section IX.

II. Module Modeling

A. Module with Variability

Our definition, as shown in **Tab.**I, of a variability-aware module is a five-tuple $(v, i, j, \Gamma, \Delta)$ mainly inspired by Christian's work [12] and the calculus follows Cardelli's module system formalization [13]. Specifically, for configuration, $o \in \mathcal{O}$ is a possible configuration option. And $\mathcal{C} = 2^{\mathcal{O}}$ contains all possible configurations ($2^{\mathcal{O}}$) that could be derived, given a configuration option can either be selected or unselected. In a module, an import or context Γ describes a general environment for a module. The import function signature is described as a continuous projection relation from configurations to function names to types as $\Gamma \in \mathcal{C} \xrightarrow{p} \mathcal{X} \xrightarrow{p} \mathcal{T}$, where $\xrightarrow{p}$ is designed to show a projection relation. The method defined in a module (Δ) can only be compiled if the configuration is available as represented as $\mathcal{C} \xrightarrow{p} \mathcal{X}$. Furthermore, this project will be furthermore projected to a set of expressions under certain types as $\mathcal{T} \times \mathcal{E}$.

For a module, it contains five components: variability module v, i is a set of configuration options that import from other modules, and j is defined in this module, import contexts Γ and functions defined Δ. We will introduce how to build a module from source in detail in Section IV-B. Besides these fundamental components in a module, we also define a constrain function $\phi_{DUC \wedge TC}\,(m)$, which will return a boolean value to evaluate whether a module is in a well-type status or not.

B. Module Constraints

Inspired by the conditional compiling in C, which is mainly represented by `#ifdef` directive, in order to include a code fragment in build-time several constraints should be satisfied and pre-checked over translation units [14]. We will introduce the constrain space with a running example.

Def-Use Constraints(DUC): Given a module, regardless its size, it should be compiled safe without any errors. In this part, we try to simulate how JVM converts source code to bytecode and reports error if necessary. Unlike JVM which really processes the code, we just record the constrains required in this step. Specifically, we recover following constraints as compiling constrains: All variables(global and local), fields, methods, classes, interfaces that are used in this module but not defined should be added as **DUC**, except those defined in third-party API or fundamental framework, like JDK. Namely, it mainly includes the constraints from the *def-use* chain.

Type Constraints(TC): A parser will return an error when it copes with a type error, which is further extended as type checker tools in variability context [15]. In type constrains, we consider the program from two perspectives:

- A type error may come from a type is used but not defined; technically, no class or interface is binding with this type;
- When using a type in a module, its parent type should also be covered. We extend this to three cases: (1) variable/field; (2) method; (3) interface/class. For example, in (1), a variable in a child class can be used without define when it is defined in its parent class; (2) is simply referred as method overriding; and (3) is considered as a case of inheritance.

More formally, as represented in Section II-A, we create a formula to represent these constraints as:

$$\phi_{DUC \wedge TC}\,(m)\,, \qquad (1)$$

which will give a boolean value to show where a module m is well-typed. Technically, we also use this function to return all missing types.

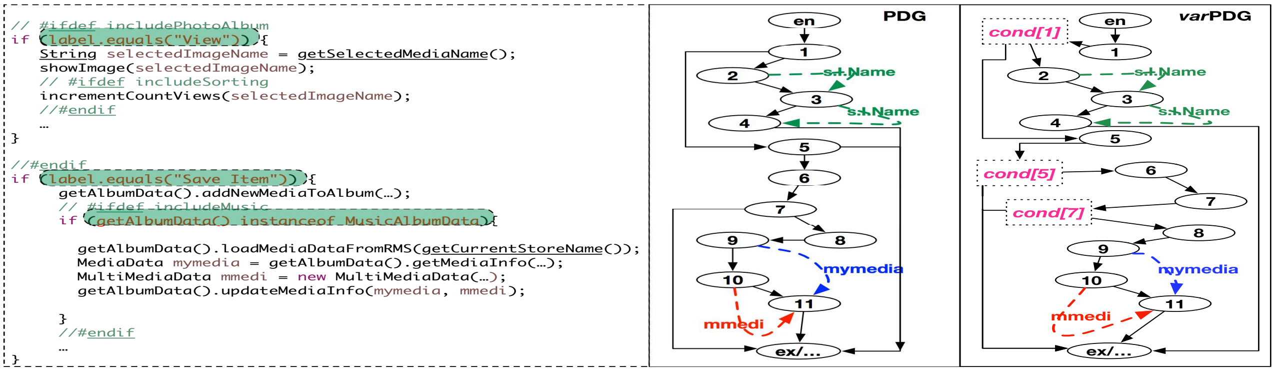
III. Variability-aware Program Dependency Graph (*varPDG*)

Unfortunately, even with the variability-aware module system, it is not sufficient to recover the feature model, considering no operations are defined on these modules. For example, what is the safe scenario to combine two modules into one, and how to map the source code to modules. In this section, we will illustrate how to give a variability-aware presentation of source code and how to explore the configuration options from the source base.

A. Building varPDG

A program dependency graph (PDG) is a combination of a program's control dependency graph (CDG) and its data dependency graph (DDG) [16]. And we extend this graph by associating conditions with method calls, which means if and only if several conditions are satisfied the method invocation or instance creation can occur. For example, considering the code excerpt shown in **Fig.**1(left), PDG is built by extracting and tracing control dependency and data dependency. To build a variability-aware PDG, namely *varPDG*, the options within the program should be tracked. In our example, there are three options at line 1, 5 and 7 respectively, and the PDG are modified by adding these options as conditions; literately, we use *cond[+]*('+' shows the line number) to show an option at a certain line.

[1] i: configuration options imported, j: configuration options defined



```
  // #ifdef includePhotoAlbum
1 if (label.equals("View")) {
2    String selectedImageName = getSelectedMediaName();
3    showImage(selectedImageName);
     // #ifdef includeSorting
4    incrementCountViews(selectedImageName);
     //#endif
     …
  }

  //#endif
5 if (label.equals("Save Item")) {
6      getAlbumData().addNewMediaToAlbum(…);
       // #ifdef includeMusic
7      if (getAlbumData() instanceof MusicAlbumData){

8        getAlbumData().loadMediaDataFromRMS(getCurrentStoreName());
9        MediaData mymedia = getAlbumData().getMediaInfo(…);
10       MultiMediaData mmedi = new MultiMediaData(…);
11       getAlbumData().updateMediaInfo(mymedia, mmedi);

       }
       //#endif
       …
  }
```

Fig. 1. An example of *varPDG*

By tracking all these options in source along with the underlying relations, the program can be represented in a variational matter, similar to variational Abstract Syntax Tree (varAST) in C [17], and labelled each node in PDG with a *presence condition* if possible. By tracking these, we can further extend our module system (in Sec.II) with *presence condition* in a variability-aware matter. Concretely, it helps to define the conditions that should be satisfied to execute some functions or code fragments. For example, recall our example in **Fig.**1, the `cond[7]` controls node 8 to 11. But there is a dependency between `cond[5]` and `cond[7]` (`cond[5]`→`cond[7]`), which leads to a result that node 8 to 11 can be executed, if and only if the condition `cond[7]`∧`cond[5]` is `TRUE`

B. Tracing Options with Pointer Analysis

To build a *varPDG*, the configuration options should be explored and extracted from source code. Unlike the macro strategy used in **C**, in OO programming language, like java, the configuration options are extremely difficult to track and explore. Considering the optional features will always be compiled or not, unlike the strategy used in **C**, where there is a stand-alone file to indicate configuration settings used in a specific build.

Listing 1. Running example of core idea in option controller

```
if(!batch){...
 if (doSplash) {
 splash = initializedSplash(); .... }
}
 ...
if(splash!=null){...}
```

1) CFL-reachability Points-to Analysis For Controlling: To resolve this, we adopt *Context-Free-Language* (CFL) reachability points-to analysis to track options with a higher precision. Here, we adopt CFL-reachability points-to resolve this [18]. Specifically, we only use language L_F and discard the regular language R_C, which ensures calling context sensitivity. L_F gives a graphical representation G of a Java program and there are four canonical statements that could define edges in between:

- Allocation `x = new O: edge` $o \xrightarrow{new} x \in G$
- Assignment `x = y: edge` $y \xrightarrow{assign} x \in G$
- Field write `x.f = y: edge` $y \xrightarrow{store(f)} x \in G$
- Field read `x = y.f: edge` $y \xrightarrow{load(f)} x \in G$

Within G, we use symbol $flowsTo \to new\,(assign)^*$ to indicate the `new` and `assign` edges. That is, `o` $flowsTo$ `v` in G represents `o` within the points-to set of `v`. Moreover, the inverse symbol $\overline{flowsTo}$ is used to trace points-to for field access. Literately, if there is an $flowsTo$ edge from `o` to `v`, there must be a $\overline{flowsTo}$ relation should from `v` to `o`. Therefore, we design algorithm **Alg.** 1 to extract statements under different cases.

Algorithm 1: Option Controller

```
   Input: $cond$
   Output: $StmtcondE$,$StmtcondUn$
1  for each $x$ $flowsTo$ $cond$ do
2      Add $x$ to $StmtcondE$ and $StmtcondUn$;
3  Add enable statements to $StmtcondE$, unable statements
   to $StmtcondUn$;
4  while $TRUE$ do
5      for $statement$ $s$ $in$ $StmtcondE$ do
6          if $there$ $is$ $a$ $c$ $flowsTo$ $or$ $\overline{flowsTo}$ $s$ then
7              Add $c$ to $StmtcondE$;
8      if $StmtcondE$ $not$ $change$ then
9          break;
10 The same process for $StmtcondUn$(line 4-9);
11 return $StmtcondE$,$StmtcondUn$;
```

Input. The input for this algorithm is condition expression *cond* in program.

Output. The outputs for this algorithm are: (1) $StmtcondE$ represents statements when this condition is satisfied; and (2) $StmtcondUn$ represent the opposite case.

Algorithm Body. For $StmtcondE$, its statements come from two aspects: one is from the statements, when the condition *cond* is enabled(line 3); and another is from points to analysis, which contains (1) the $flowTo$ relation ends with $code$ and (2) iteratively adding the statements having $flowsTo$ or $\overline{flowsTo}$ relation to the statements in $StmtcondE$ until the set does not change (line 4-9). And we apply a similar process for $StmtcondUn$.

Example. As the code segment shown in **List.**1, the option controller algorithm will extract three options: `batch`, `doSplash` and `splash`. And it can find the underlying relations between these options as: ¬ `batch` ∧ `doSplash` → `splash`, which mean the option `splash` is dependence on both ¬ `batch` and `doSplash` are `TRUE`.

IV. VMS Feature Model Recovery Approach

A. Overview

Now, we will introduce our idea in general. Basically, building a feature model from a legacy source requires processing source code and clustering similar code fragments into several clusters. However, this faces the obstacle when coping with certain cases (see Section I **motivation**). Our approach, variability-aware feature model recovery, *VMS*, will build the variability-aware modules to resolve this. In general, it contains three steps:

1) **Step 1**: Initially, *VMS* will extract all common modules from the system. For example, in **Fig.**2, *VMS* will first find the source code for `SPL` and `PrevalyerSPL` without any optional features;
2) **Step 2**: Then, *VMS* will enable one configuration option in the common modules found in **Step 1** to track all optional features. For each iteration, if there are some modules that become valid during this iteration, we will try to package these modules into a cluster, which is an *optional feature* in our context. In the example, we try to find *optional feature* like `Replication` along with its configuration option; and
3) **Step 3**: For each potential *optional feature* found in **Step 2**, we perform a type-checking and concern separation checking to adjust the modules in each feature if necessary. For example, this step will check whether the *feature* `Replication` is well-typed and whether there is a need to divide it into two subfeatures. And we also do this checking on those common features found in **Step 1**.

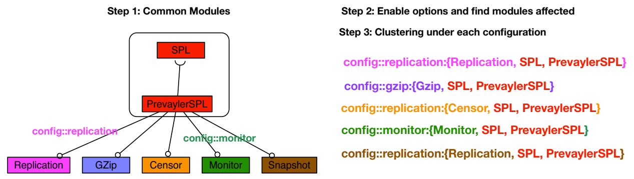

Step 1: Common Modules
SPL
PrevaylerSPL
config::replication
config::monitor
Replication
GZip
Censor
Monitor
Snapshot
Step 2: Enable options and find modules affected
Step 3: Clustering under each configuration
config::replication:{Replication, SPL, PrevaylerSPL}
config::gzip:{Gzip, SPL, PrevaylerSPL}
config::replication:{Censor, SPL, PrevaylerSPL}
config::monitor:{Monitor, SPL, PrevaylerSPL}
config::replication:{Replication, SPL, PrevaylerSPL}

Fig. 2. The core idea of our VMS approach

B. Build Module from Source

First, we will broadly introduce how to construct a module from the source base. In this paper, we adopt Java as a target language for our case study and the approach defined in this paper could easily extend to other Object-oriented environment. We build a module for each class or interface in a target program P and group the programming elements parsed by following syntax. Specifically, in the syntax, we give an ASTNode expression on the mapping from source code to terms defined in our variability-aware module system. And all ASTNodes' types defined under Eclipse Java development tools (JDT) [2].

TABLE II
Syntax of VMS

Notation	Syntax	Remark	Subremark
$m \in \mathcal{M}$	::=	module	
	$v \in \mathcal{V}$		varability model
	$i \in \mathcal{O}$		import option
	$j \in \mathcal{O}$		export optio
	Γ		import func. sig.
	Δ		func. definition
$e \in \mathcal{E}$	::=	expression	
	`Exp.`		Expression
$o \in \mathcal{O}$	::=	configuration option	
	`pt(Do)`		
	`pt(For)`		
	`pt(If)`		
	`pt(While)`		
	`pt(Switch)`		
Δ	::=	function definition	
	`MethDecl.`		
$t \in \mathcal{T}$	::=	types	
	`ITypeBind.`		

Specifically, within a module m, it contains v for variability model, i for import configuration option, j for export option, Γ for import function signatures and Δ for functions defined in module. The expression $e \in \mathcal{E}$ in a module could be extracted from ASTNode `Expression`. And the option $o \in \mathcal{O}$ in a module could be obtained by applying points-to analysis with function $pt(A)$, where $pt(A)$ returns all pointers defined in the conditional expression of A. In detail, it will check all ASTNodes that can lead to branches, including `DoStatement`, `ForStatement`, `EnhancedForStatement`, `IfStatement`, `WhileStatement`, and `SwitchStatement`. Here, we define the configuration option as the conditional expression in Java, due to following reasons: (1) a conditional expression can lead to a certain path in the control flow graph with a specific context. That means a conditional expression could give a variability context for configuration according to several empirical studies [19], [20]; and (2) our points-to analysis gives a comprehensive understanding of underlying relations between these options. The type $t \in \mathcal{T}$ could be obtained using type-resolving techniques provided by Eclipse with an input of the binding of the type. As for the function definition Δ, it can be extracted by visiting all `MethodDeclaration` node in the AST. The rest of components in module m, including $i, j \in \mathcal{O}$, Γ and $v \in \mathcal{V}$, could be built based on these basic elements using our definition in Section II-A.

[2] Eclipse JDT: http://www.eclipse.org/jdt/

C. Module to Feature

Moving on, we will cluster these modules into features, where a feature is composed of one or more modules. We first introduce the measurement for computing the distance between a module and a feature.

1) Topology based Method Reference: Adopted from Robillard's topology work [8], we adjust it to compute the uniqueness of a module to a feature. The core idea of topology analysis is it computes the similarity using two metrics *specificity* and *reinforcement*. Specifically, the *specificity* suggests that if an element A only refers to an other element B should be ranked higher comparing to C refer to many elements including B. And the intuition behind *reinforcement* is that if elements refer to (or referred from) many elements are in one cluster, possibly they should be considered as a part of that cluster. Therefore, we simply compute the uniqueness from a module m to a feature f as follows.

$$w_{tmr}\left(m,f\right)=\frac{1+\left|targets(m)\cap f\right|}{\left|targets\left(m\right)\right|}\cdot\frac{\left|sources(m)\cap f\right|}{\left|sources\left(m\right)\right|},\tag{2}$$

where $targets(m)=\{m'\left|(m,m')\in R\right\}$ and $sources(m)=\{m'\left|(m',m)\in R\right\}$. Here $(m',m)\in R$ represent there is method invocation starts from m' and ends with m.

2) Type Reference: The type reference gives an overview on types need to be resolved for a module to be a well-typed module. The type reference ensures consistency and type-safe and works at a fine granularity. The underlying idea in type reference is that for each module, it looks up all possible references, such as, method reference - from method invocation to method definition; variable reference - from variable access to its definition; or type reference - from a type reference to its declaration and explore the types referred in all these references. For example, in module m, a method defined in type t is invoked, then we add the reference from module m to type t. For the type reference, we define two types of vectors: $def\left(X\right)$ and $ref\left(X\right)$. Specifically, the vector $def\left(X\right)\in\mathbb{R}^n$ defines all types within X with n represent the number of types in the subject program. $def\left(X\right)=[d_1,...,d_n]$ is defined as if type i is defined in X, then d_i should be 1, otherwise d_i is 0. On the contrast, $ref\left(X\right)=[r_1,...,r_n]$ shows the reference information. If a type i is referenced by X, then the ith element r_i in $ref\left(X\right)$ should be 1, otherwise it should be 0. Therefore, the type reference distance w_{tr} is defined as follows.

$$w_{tr}\left(m,f\right)=\frac{1}{2n}\left(\sum_{m_i\in f}\left(\begin{array}{c}csd\left(def\left(m\right),ref\left(m_i\right)\right)+\\csd\left(ref\left(m\right),def\left(m_i\right)\right)\end{array}\right)\right),\tag{3}$$

where n is the number of modules currently in feature f and csd is defined as cosine similarity between two vectors:

$$csd\left(X,Y\right)=\frac{X\cdot Y}{\left\|X\right\|\left\|Y\right\|}.$$

The subtlety of this approach is it uses the cross reference to check how a module m_i in feature f relies on m with $csd(def(m),ref(m_i))$ and the opposite case showing how the module m relies on a module m_i. Here, we exclude the type defined outside of this program, like a type defined in a third-party API or a type defined in Java runtime environment.

3) Documental Topic Similarity: In a broader sense, a program can be considered as a set of *documents* and defined as a *corpus*. Upon this *corpus*, an information retrieval approach named Latent Dirichlet Allocation (LDA) [21] can be used to extract the topic distribution. Furthermore, a topic z is given based on a multinomial probability distribution upon a set of words ws obtained from a Dirichlet distribution with the shape parameter β. For example, given a topic label "life" and relative words "biology", "gene", "water", and "oxygen" can be represented with a certain probabilities, which is learned from the *corpus*. Using LDA allows us to build a bridge between a module and a feature. That is, for each module, a vector is defined with each item infer the probability on each topic. For example, if there are 5 topics and a module m is represented as a topic distribution $\theta_m=[0.5,0.3,0.1,0.9,0.1]$, which states that this module should be considered a part of topic with $id=4$ with probability 0.9 from a textual perspective. Then some similarity measure could be used to measure the distance, like cosine distance or Kullback-Leibler. Technically, in this paper, we use MALLET tool to create the topic model and adopt an empirical setting for parameter in LDA with $\alpha=50/T$ and $\beta=0.01$, since it has shown its strength across different corpora [21]. Therefore, we define the topic similarity w_{dt} using cosine similarity as:

$$w_{dt}\left(m,f\right)=\frac{1}{n}\sum_{m_i\in f}\frac{\theta_{m_i}\cdot\theta_m}{\left\|\theta_{m_i}\right\|\left\|\theta_m\right\|},\tag{4}$$

where θ_m defines the topic distribution of m and n represents the total number of modules in feature f.

4) Put All Pieces Together: For all distance values extracted (w_{tmr}, w_{tr} and w_{dt}), we define an overview distance value as w_*. The overview distance w_* is defined by following Robillard's approach, which uses operator $x\uplus y=x+y-x\cdot y$ to combine two values [8]. This operator yields a result by equally treating all arguments and return its overall result within the range $[0,1]$. Thereby, we calculate the overview distance by: $w_*=w_{tmr}\uplus w_{dt}\uplus w_{ss}$. In addition, we have to put the variability-aware constraints on w_*. The value of w_* represent the overall similarity between a module and a feature; the more w_* close to 1, the more similar they are.

However, combining two modules with high similarity may still introduce errors, which may come from: (1) import functions signature merging ($\Gamma_x\left(c\right)\cup\Gamma_y\left(c\right)$) under a configuration $c\in\mathcal{C}$, (2) combining variability modules ($v_x\cup v_y$) and (3) merging imported and self-defined configuration options in each module ($i_x\cup i_y$ and $j_x\cup j_y$). Therefore, to merge two compatible modules to yield a new one, the conflict should be checked to ensure all modules are well-formed. The conflict checking process is shown by following logical proofing.

[2] MALLET: http://mallet.cs.umass.edu/

$$\frac{\begin{array}{c} m' = (v', i', j', \Gamma', \Delta') \\ v' = v_x \cap v_y \setminus conflict\left(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y\right) \\ \Gamma'\left(c\right) = \Gamma_x\left(c\right) \cup \Gamma_y\left(c\right) \backslash \left(sig\left(\Delta_x\left(c\right)\right) \cup sig\left(\Delta_y\left(c\right)\right)\right) \\ \Delta'\left(c\right) = \Delta_x\left(c\right) \cup \Delta_y\left(c\right) \\ i' = i_x \cup i_y \backslash \left(j_x \cup j_y\right), j' = j_x \cup j_y \end{array}}{\left(v_x, i_x, j_x, \Gamma_x, \Delta_x\right) \bullet \left(v_y, i_y, j_y, \Gamma_y, \Delta_y\right) = m'}$$

The module checking, specifically the type checking, of combining option $\bullet$ detect the error in all components for each module. Specifically, the module checker will check: (1) the conflicts from two modules' merged variability model v' ($v' = v_x \cap v_y \setminus conflict\left(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y\right)$), which mean this conflict should merge v_x and v_y with excluding the conflict from configurations. This exclusion of configuration ensures that the new variability module v' can fully map the functions defined within the new module m' and all imports. This checks for the conflict from both modules import functions with same name but different types; (2) defining sig gives a mapping $sig : (\mathcal{X} \rightarrow \mathcal{E} \times \mathcal{T}) \rightarrow (\mathcal{X} \rightarrow \mathcal{T})$. Therefore $sig\left(\Delta\right)$ returns a mapping $\mathcal{X} \rightarrow \mathcal{T}$, which is a Γ in our definition. Therefore, for the merged import function $\Gamma'(c)$ under the configuration c with exclusion of functions required by one module but defined in another module; (3) merge the configuration-option defined $j' = j_x \cup j_y$ and functions defined $\Delta'\left(c\right) = \Delta'_x\left(c\right) \cup \Delta'_y\left(c\right)$; and (4) merge the configuration-option imports and remove the import configuration option from one module, but already defined in another $i' = i_x \cup i_y \setminus \left(j_x \cup j_y\right)$. Therefore, we adopt this checking to check whether a module merge action is allowed or not. Formally, we define a function to do this checking as follows.

$$\phi\left(m_x \bullet m_y\right). \tag{5}$$

This function will return a boolean value to show whether this merge will lead to any conflict. Return TRUE, if this merge is safe, and FALSE for the unsafe merge.

Algorithm 2: *VMS* feature model constructing approach

```
   Input: $\#op$,$\#cf$,$P$
   Output: $fm$
 1 Build the $varPDG$ for input program $P$;
 2 Create an empty module to class/interface mapping $D$;
 3 for class $c$ in $P$ do
 4     create a module m using c;
 5     add $(m, c)$ to $D$;
 6 Create set $CommonM$;
 7 Create empty queue $Q$, add entry class $en$'s main function
   $main$ to $Q$;
   /* find common modules                                      */
 8 while $Q$ not empty do
 9     $f_{head} \leftarrow Q.poll$;
10     if $f_{head}$ not visited then
11         Find all functions $needprocess$, which not defined
           in the scope of a condition $cond$ in $varPDG$;
12         All all modules contain $needprocess$ to
           $CommonM$;
   /* hierarchical clustering all optional module
      set and common module set respectively                   */
13 For each module $commonM_m$ in $commonM$ into a
   cluster $commonM_i$;
14 Let optional modules $optionalM$ be
   $optionalM \leftarrow D.keyset() \setminus commonM$;
15 while $commonM.size > \#cf$ do
16     Find two cluster $commonM_i$ and $commonM_j$ with
       maximum $w_*\left(commonM_i, commonM_j\right)$ and
       $(commonM_i \cdot commonM_j, OK)$;
17     Update all reference information;
18 while $optionalM.size > \#cf$ do
19     Find two cluster $optionalM_i$ and $optionalM_j$ with
       maximum $w_*\left(optionalM_i, optionalM_j\right)$ and
       $(optionalM_i \cdot optionalM_j, OK)$;
20     Update all reference information;
21 Create $fm$ from cluster recovered;
22 return $fm$;
```

D. VMS

Our ultimate target is to build a feature model by analyzing the source base.

Input. The input for *VMS* approach includes two values: $\#op$ represents the number of option features preferred in this feature model, and $\#cf$ shows the number of common features preferred.

Output. The output returns the feature model fm proposed by *VMS* automatically.

Algorithm Body. Now, we will introduce the main process of *VMS* approach to explore the feature model for source code. Due to the length of this algorithm, we separate it into several sections and introduce them respectively.

- Line 1 - 6: First, extract structural information to build $varPDG$ and for each class/interface create a module;
- Line 7 - 12: We define the common modules as all modules that must be executed regardless of input context. Therefore, we follow a Breadth-First-Search structure to add the "must" invoked functions into the queue Q iteratively. To find these "must" invoke functions, *VMS* discards method invocation enclosed in statements under a certain configuration option $op \in \mathcal{O}$. With that, the modules associated with these functions are considered as common modules;
- Line 13 - 14: As illustrated in the core idea(see **Section.**IV-A), we create two sets: one contains all modules for common features $commonM$ and another contains all modules for optional features $optionalM$;
- Line 15 - 17: We conduct a hierarchical clustering on the common module set $commonM$ based on our distance measurement w_* under a module conflict checking formula (X, Y, OK). Here, (X, Y, OK) represents there is no module conflict between module X and Y; Specifically,

it will do the checking from two parts: (1) check the module constraints and (2) check potential errors for variability merging. Thereby, the conflict checking formula could be represented by.

$$(X, Y, OK) = \phi_{DUC\wedge TC}(X) \wedge \phi_{DUC\wedge TC}(Y) \wedge \phi(X \bullet Y)$$

- Line 18 - 22: We also apply the hierarchical clustering upon all modules belonging to the optional module set $optionM$.

V. Case Study

A. Experimental Settings

We will introduce the infrastructure we selected, how we do constraints checking in terms of type and linker constraints mentioned, and the ground truth for assessing the performance.

1) Infrastructure and Constraints Checking: To implement our work, we develop a Eclipse plugin system and integrate with `TypeChef` system. The `TypeChef` system is a variability-aware parsing tool, which gives a customized compiling service and variability-aware presentation for code fragments [15]. `TypeChef`[3] is used to provide the constrain checking for our module systems. Unlike the normal use of `TypeChef`, which gives the type error during the checking, in our work, we only need it to return types required to be resolved at a certain point.

2) Ground Truth for Performance Assessment: In order to assess the performance of our work and compare with our tools, we have to select the systems that have been well-researched with two kinds of information available publicly: (1) given our final target is to build a feature model from source base, the feature model should be available which could be used to assess whether our approach could extract correct feature relations. For example, if there is an *implies* relation from a feature f to feature f', ideally a competitive approach should give this relation as a part of output; and (2) some information should be available that tells how code fragments and features are mapped.

B. Subject Systems

Based on the rule for performance assessment, we carefully select subject systems from different domains to verify our approach in multiple dimensions. A special factor to consider is the specific type of subject system, namely we have to test our approach using systems from different domains and in different size scales, including small systems, medium-size systems and large scale systems. For subject systems, we mainly target on systems in Java, systems in `C` and `CPP` are out of the scope of this paper, since they use directives to implement variability and associate with configuration management techniques. Therefore, following systems were selected for our study.

- **Prevayler**[4]. An open-source object persistence library for Java with 8009 LOC. It is a well recognized product for product line research [22], [23], 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 v8** . Originally developed by University of Lancester, UK as a product line with 4653 LOC [24]. It contains several features: *Photo, Music, SMS Transfer, Copy Media, Favourites*, and *Sorting*. The dependencies include: $Photo \vee Music$, $SMSTransfer \Rightarrow Photo$ and

$$MediaTransfer \Leftrightarrow (SMS\,Transfer \vee Copy\,Media)\,.$$

- **ArgoUML**[5] with 120 KLOC, provides modeling support for UML v1.4 diagrams and supports multiple programming languages. In ArgoUML, following seven features are selected: *Cognitive*, *Activity Diagram*, *StateDiagram*, *Collaboration Diagram*, *Sequence Diagram*, *Use Case Diagram*, *Deployment Diagram* from an empirical research [25]. The feature *Logging* is not covered in our mining work, as it is not a callable feature for end customers. In another study [22], a dependency $ActivityDiagram \Rightarrow StateDiagram$ is added. In our experiment, we adopt this setting.
- **Berkeley DB (in Java)**[6] is a database application with 84KLOC, and 38 features. Berkeley DB could be an embedded application to other applications and provides a storage engine. It performs safety transaction and several useful APIs to cope with IO, logging, memory and so forth. A full feature list and feature relation is enclosed in our project webpage[7].

C. Tools

We have implemented a prototype of our work and integrated other relative approaches into an Eclipse plug-in named *LoongFMR*. We have released the experimental data, ground truth, and source code on our project page: http://www.chrisyttang.org/loong_fmr/.

VI. Experimental Result

A. Relative Approaches

1) ACDC: Algorithm for Comprehension-Driven Clustering (ACDC) recovers the architecture of system by inspecting certain patterns that could exist in systems [26]. Specifically, ACDC contains *source file pattern*, *body-header pattern*, *leaf collection and support library pattern*, and *ordered and limited subgraph domination*. ACDC identifies clusters by using these patterns with orphan adoption techniques, which originally proposed in [27].

2) LIMBO: scaLable InforMation BOttleneck (LIMBO) optimizes the usage of information loss when conducting clustering on a system. It builds on *Information Bottleneck (IB)* framework and could collected relevant information during clustering [28].

[3] TypeChef is designed in C, a Java version of TypeChef's core function is implemented as a part of LEADT tool, available at: https://github.com/ckaestne/LEADT

[4] Prevayler: available at http://prevayler.org

[5] ArgoUML: available at: http://argouml.tigris.org

[6] Berkeley DB: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/

[7] LoongFMR: http://www.chrisyttang.org/loong_fmr/

3) ARC: Architecture Recovery using Concern (ARC) as defined in [29] uses a generative probabilistic model for text corpora named Latent Dirichlet Allocation (LDA) to retrieve concerns and identify programming elements belonging to concerns. It treats the source as a series of *documents* that contains various topics and then measures the similarity using the Jensen-Shannon divergence (D_{js}).

4) Bunch: Bunch regards the recovery task as an optimization program [30]. It starts with a random partition and iteratively updates each cluster by optimizing the object function called Modularization Quality (MQ) until it cannot find a better solution.

5) W-UE and W-UENM: Weighted Combined Algorithm(WCA) combines hierarchical clusters into larger sets by computing the inter distance between two possible variants using Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM(UENM) distance measurement respectively [31].

B. Metrics

We measure the performance of different approaches by four aspects. We adopt three metrics from software architecture recovery: *MoJo similarity*,*architecture-to-architecture measurement* and *cluster-to-cluster coverage*. The runtime performance returns the execution speed of the algorithm.

1) MoJo Similarity: SimilarMoJo metric [32] gives a representation of closeness between two architectures with a percentage. It helps to analyze two different architecture strategies. SimilarMoJo is defined as:

$$SimilarMoJo\left(A,B\right) = \left(1 - \frac{MoJo\left(A,B\right)}{N}\right) \times 100\%, \tag{6}$$

where $MoJo(A,B) = min\left(mno\left(A,B\right), mno\left(B,A\right)\right)$ and $mno\left(A,B\right)$ represents the minimum number of *Move* or *Join* operations needed to transform from A to B or vice versa. N represents the number of units in the system. The algorithm in [33] gives a way to calculate $mno\left(A,B\right)$. Furthermore, the symbol $\forall A$ in the denominator represents a partition of A and $\max\left(mno\left(\forall A,B\right)\right)$ means the maximal distance from any partition A to B.

TABLE III
EVALUATED MOJO SIMILARITY MEASURING

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	83.0	75.0	63.69	83.93
LIMBO	55.56	68.75	52.03	78.31
Bunch	74.07	68.42	63.92	88.77
ARC	59.25	73.43	53.09	84.95
VMS	83.33	71.87	79.78	81.62
W-UE	66.67	78.12	51.66	83.41
W-UENM	68.52	71.87	51.66	82.14

2) Architecture-to-architecture Measurement(a2a): $a2a$ is developed to overcome the limitation of *MoJo* measuring discrepancy of files between the recovered result and ground truth [34]. *a2a* measures two architectures, one is the recovered and another is ground truth by computing:

$$a2a\left(A_i,A_j\right) = \left(1 - \frac{mto\left(A_i,A_j\right)}{aco\left(A_i\right) + aco\left(A_j\right)}\right) \times 100\% \tag{7}$$

$mto\left(A_i,A_j\right) = remC\left(A_i,A_j\right) + addC\left(A_i,A_j\right) + remE\left(A_i,A_j\right) + addE\left(A_i,A_j\right) + movE\left(A_i,A_j\right)$ and $aco\left(A_i\right) = addC\left(A,A_i\right) + addE\left(A,A_i\right) + movE\left(A,A_i\right)$, where the symbol $mto\left(A_i,A_j\right)$ is the minimum changes from architecture A_i to A_j and $aco\left(A_i\right)$ represents the total number of operations from a "null" architecture A into A_i. There are five operations that could be used to transform an architecture to another including: additions(*addE*), removals(*remE*), and moves(*movE*) from one cluster to another.

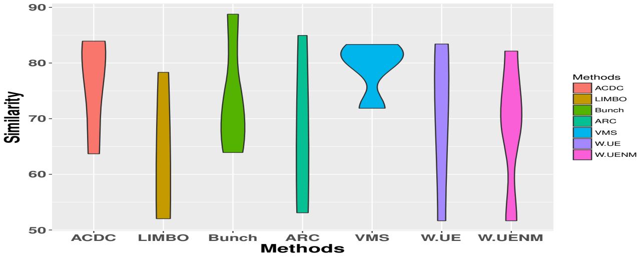


Fig. 3. The violin plot for MoJo Similarity

TABLE IV
EVALUATED PROJECT AND ARCHITECTURE ON A2A MEASURING

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	21.18	30.31	51.46	17.97
LIMBO	47.19	63.68	47.67	55.57
Bunch	45.07	56.75	49.06	49.08
ARC	49.23	67.39	49.76	63.48
VMS	52.16	67.70	52.87	62.92
W-UE	50.0	73.56	49.90	62.01
W-UENM	50.0	73.56	49.90	61.64

The results shown in **Tab.**III, **Tab.**IV, and associate violin plots, including **Fig.**3 and **Fig.**5, indicate that at the system level[8], our *VMS* approach could reach a competitive result and more importantly the result is stable comparing to others. From these two violin plots, we can draw the following conclusions: (1) the median value from *VMS* gives a better performance than others; and (2) from the distribution and range between first and third quartile, our *VMS* shows its strength in providing stable results.

3) Cluster-to-cluster Coverage($c2c_{cvg}$): $c2c_{cvg}$ explores the component-level accuracy and is given as [34]:

$$c2c\left(c_i,c_j\right) = \frac{\left|entities\left(c_i\right) \cap entities\left(c_j\right)\right|}{\max\left(\left|entities\left(c_i\right)\right|, \left|entities\left(c_j\right)\right|\right)} \times 100\%, \tag{8}$$

where c_i is a cluster generated by clustering techniques and c_j is the cluster from the ground-truth. The $entities(c)$ shows all candidates in the cluster c. The *architecture coverage* $c2c_{cvg}$ is a metric that extend the clusters overlap as:$c2c_{cvg}\left(c_1,c_2\right) = \frac{|simC(A_1,A_2)|}{|A_2.C|} \times 100\%$, where $simC\left(A_1,A_2\right) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge (c2c\left(c_i,c_j\right) > th_{cvg})\}$. A_1 is the recovered architecture; on

[8] Metrics MoJo and arch2arch give a system-level assessment, and cluster2cluster coverage returns a cluster-level assessment

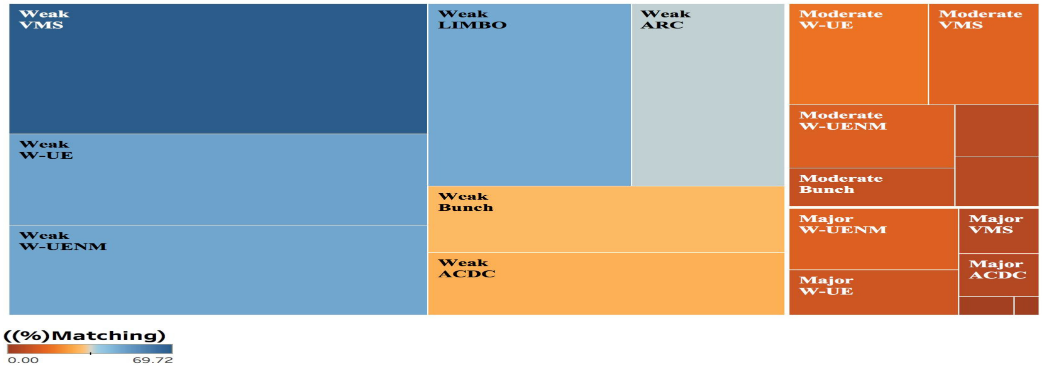


Fig. 4. The Heat Map for Cluster-to-cluster Coverage

TABLE V
CLUSTER-TO-CLUSTER MEASURING(MAJORITY MATCH(50%), MODERATE MATCH(33%),WEAK MATCH(10%))

Algorithm	Prevayler			MobileMedia			ArgoUML			BereleyDB		
	Major	Mod.	Weak	Major	Mod.	Weak	Major	Mod.	Weak	Major	Mod.	Weak
ACDC	13.64	18.18	54.55	0.00	0.00	30.00	4.55	4.55	22.73	0.00	0.00	7.32
LIMBO	0.00	0.00	80.00	0.00	0.00	28.57	0.00	0.00	66.67	0.00	0.00	14.63
Bunch	5.26	15.79	47.37	0.00	14.29	42.86	0.00	3.03	21.21	0.00	0.00	9.76
ARC	0.00	0.00	20.00	0.00	14.29	57.14	0.00	0.00	22.22	2.38	7.14	45.24
VMS	16.67	16.67	83.33	0.00	14.29	71.43	0.00	22.23	77.78	2.44	4.88	46.34
W-UE	40.00	40.00	60.00	0.00	28.57	71.43	0.00	0.00	11.11	0.00	4.17	54.17
W-UENM	40.00	40.00	60.00	14.29	14.29	71.43	0.00	0.00	11.11	0.00	0.00	50.00

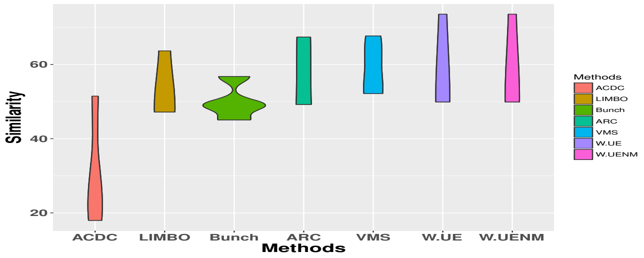


Fig. 5. The violin plot for a2a Measurement

the contrast, A_2 is architecture from the ground-truth. The symbol $A_2.C$ represents all clusters in A_2, and th_{cvg} shows the threshold that indicates the bottomline for clustering approach must achieve in order to count for similar clustering when comparing to A_2. The detail performance of our approach and other relative approaches are shown in **Tab.**V. In addition, the heat map in **Fig.**4 indicates *VMS* returns considerable results in terms of weak match($> 10\%$) and majority match($> 50\%$).

4) Run-time Performance: Run-time performance explores the execution time under the same environment. In this paper, all algorithms are run on a MacOS 10.12 with Intel i5 2.6GHz, 8G 1600 MHz DDR3, and targeting on Eclipse 4.5 with JRE 7.

TABLE VI
RUNTIME PERFORMANCE IN MILLISECOND

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	665	215	46127	1394
LIMBO	555	165	2629637	1349
Bunch	192	229	25284	258
ARC	2356	3247	139883	6408
VMS	122	313	453278	21847
W-UE	235	67	30309	733
W-UENM	144	50	32054	551

As the runtime performance presented in **Tab.**VI, a potential bottleneck for *VMS* is it requires more resource when processing large scale systems, which might due to the conflict checking and also computing the complicated model (w_*) to measure module similarity.

VII. DISCUSSION

A. Lessons Learned

In this section, we will describe experience learned from this study and share several empirical understandings in feature model building by answering following research questions. By answering these questions, the strength and potential weakness of *VMS* are presented.

RQ1*: Is current architecture recovery technique qualified for constructing feature model?*

As the results shown in the previous section, we can conclude that traditional techniques designed for architecture recovery cannot meet the need of feature model construction. Another apparent limitation for other approaches is that they cannot ensure all programming elements in a cluster are well-typed, which is solved in our approach. Therefore, our strategy could be a better choice for product line feature model building.

***RQ2**: What are the potential limitations for VMS approach?*

Although *VMS* approach returns a competitive result on four case studies, it still has its limitations. The limitations are two parts: (1) the first limitation is from runtime performance as we described in previous section; and (2) another limitation is that it is still a coarse-granularity approach. Since, after we carefully check the ground-truth, we found that some programming elements are shared by different features. This can only be resolved using a fine-granularity strategy. Whereas, given the goal of building feature model, a fine-granularity work might be overfit considering we only need to build a feature model to provide a raw view of the system, for which our approach is fully qualified.

B. Threats to Validity

Construct and Internal Validity. The metrics, including SimilarMoJo, cluster-to-Cluster, and achitecture-to-architecture are broadly adopted in architecture recovery performance collection and have been tested on various target systems. The benchmark are collected from other researchers' work, which are theoretically acceptable. Nevertheless, they may be incorrect as there is a widely recognized truth that there is no single "correct" architecture.

External Validity. (1) Even the size of our subject systems includes two small systems (4K, 8KLOC), a medium-size system (84KLOC) and a large-scale system (120KLOC) , due to the number of cases, the experimental results are not intend to be generalized to all systems. This is mainly because we have to restrict the systems to those with ground truth available; (2) Further in the assessment, we adopt the common architecture recovery performance metrics to testify our approach to reduce the bias of using self-defined approach. Clearly, the correctness of ground truth can highly influence the performance.

VIII. Related Work

Feature model recovery for product line, technically, is highly related to software architecture recovery and variability modeling.

A. Feature Model Recovery Techniques

For feature model recovering, current work mainly focuses on following directions: (1) recovery feature model by analyzing all products within the product family [35], [36]. Specifically, (2) recover the feature model by tracking the version change [37]; (3) recover the feature model from requirement specifications [38], [39]; and (4) other works [40], [41] requires multiple input, like source code, requirement specification, even architecture information, rather than starting from a pure code base. For example,She's work [40] build the feature model by identifying parent candidates for the given feature. Our approach is different from She' work in two fold:(1) She's work requires a list of feature names with detailed description; (2) to support the dependencies, it also need a propositional formula to represent this. Whereas, in our approach, rather than relying on the feature list, we need the number of common features and count of optional features. For dependencies and constraints, we detect and explore these by analyzing the architecture and building modules. Our approach differs from these approaches in terms of different input even if we have the same targets. Our limit our work to Java, since different languages provide different mechanism for variability implementation. For example, in C, people prefer preprocessor and, in C++, template is a widely adopted choice [42].

B. Software Architecture Recovery Techniques

Software Architecture Recovery(SAR) techniques have been broadly used to rebuild the product architecture by collecting syntax and structural information from the system [26], [29], [30], [43], [44]. **Granularity.** Most architecture recovery techniques are implemented at a class and file level, which means they parse a single class file as an unit for recovery [30], [44], [45]. **Methodology.** For the target of reconstructing or learning architecture, some approaches deem it as an optimization problem [30], [44], [46], in which some objective functions, like modularity, fan-in, fan-out, are built and the architecture that possible satisfy and offer maximal or minimal target values are considered as solutions; some representative works use searching patterns along with users' instructions to find target patterns and rebuild the architecture by sequentially finding these components [47], [48]. In addition, adopting clustering algorithms to resolve architecture problem is another trend, since programming components, like class, with similar function should be grouped into the same cluster [28], [31]. Another direction to achieve the target is textual information, namely natural language processing technique, like the strategies shown in [29], [43]. Among all approaches, we mainly give a comparison with several representative approaches used in architecture recovery, including **ACDC**, **LIMBO**, **Bunch**, **ARC** and **W-UE(NM)** as the detail shown in section VI.

IX. Conclusions And Future Work

Constructing feature model and modeling variability are promising and worth investigating in product-line oriented research. In this paper, we proposed an approach on constructing feature model by investigating variability-aware modules. As our results suggest, traditional methods used in architecture recovery could not reach a stable and competitive performance comparing to our variability-aware approach. In the future, we will try to extend this work by tracing users' behaviors during program execution and build a mapping between users' behaviors and source code.

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

[3] C. Krueger, "Eliminating the adoption barrier," *IEEE Softw.*, vol. 19, no. 4, pp. 29–31, 2002.

[4] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Clelang-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," 2013.

[5] Al-Msie'Deen and Ra'Fat, "Mining feature models from the object-oriented source code of a collection of software product variants," 2013.

[6] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[7] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," pp. 69–78, 2015.

[8] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 18:1–18:36, Aug. 2008.

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

[10] 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.

[11] 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.

[12] C. Kästner, K. Ostermann, and S. Erdweg, "A variability-aware module system," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 773–792. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384673

[13] L. Cardelli, "Program fragments, linking, and modularization," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97. New York, NY, USA: ACM, 1997, pp. 266–277. [Online]. Available: http://doi.acm.org/10.1145/263699.263735

[14] S. Nadi, T. Berger, C. KÃd'stner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, Aug 2015.

[15] 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," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 805–824. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048128

[16] 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, Jan. 2005.

[17] 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. 14:1–14:39, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2211616.2211617

[18] G. Xu, A. Rountev, and M. Sridharan, "Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 98–122. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_6

[19] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden, "Variational data structures: Exploring tradeoffs in computing with variability," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661143

[20] M. Erwig and E. Walkingshaw, "The choice calculus: A representation for software variation," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 6:1–6:27, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063239.2063245

[21] M. Steyvers and T. Griffiths, "Probabilistic topic models," *Handbook of latent semantic analysis*, vol. 427, no. 7, pp. 424–440, 2007.

[22] 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.

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

[24] 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.

[25] 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.

[26] V. Tzerpos and R. C. Holt, "Acdc: An algorithm for comprehension-driven clustering," p. 258, 2000.

[27] V. Tzerpo, "The orphan adoption problem in architecture maintenance," in *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, ser. WCRE '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 76–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832304.836986

[28] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[29] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 552–555. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100123

[30] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999, pp. 50–59.

[31] O. Maqbool and H. A. Babri, "The weighted combined algorithm: a linkage algorithm for software clustering," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, March 2004, pp. 15–24.

[32] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, June 2004, pp. 194–203.

[33] W. Zhihua and V. Tzerpos, "An optimal algorithm for mojo distance," in *Program Comprehension, 2003. 11th IEEE International Workshop on*, May 2003, pp. 227–235.

[34] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 235–245.

[35] Y. Xue, "Reengineering legacy software products into software product line based on automatic variability analysis," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 1114–1117.

[36] Y. Yang, X. Peng, and W. Zhao, "Domain feature model recovery from multiple applications using data access semantics and formal concept analysis," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 215–224.

[37] Y. Xue, Z. Xing, and S. Jarzabek, "Understanding feature evolution in a family of product variants," in *2010 17th Working Conference on Reverse Engineering*, 2010, pp. 109–118.

[38] M. G. R. Stoiber, "Modeling and managing tacit product line requirements knowledge," in *Managing Requirements Knowledge (MARK), 2009 Second International Workshop on*, Sept 2009, pp. 60–64.

[39] S. Buhne, K. Lauenroth, and K. Pohl, "Modelling requirements variability across product lines," in *13th IEEE International Conference on Requirements Engineering (RE'05)*, Aug 2005, pp. 41–50.

[40] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985856

[41] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse Engineering Architectural Feature Models," in *5th European Conference of Software Architecture (ECSA)*, Springer, Ed., vol. 6983. Essen, Germany: Springer, Sep. 2011, pp. 220–235. [Online]. Available: https://hal.inria.fr/inria-00614984

[42] K. Czarnecki and E. Ulrich, *Generative Programming: Methods, Tools, and Applications*. Reading, MA, USA: Addison-Wesley, 2000.

[43] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, March 2011, pp. 35–44.

[44] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 462–471.

[45] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[46] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, March 2011.

[47] S. Kamran, "Alborz: a query-based tool for software architecture recovery," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, 2001, pp. 115–116.

[48] K. Sartipi, "Software architecture recovery based on pattern matching," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept 2003, pp. 293–296.