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 diagram 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 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
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.1, 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, \(o \in O\) is a possible configuration option. And \(C = 2^O\) contains all possible configurations \((2^O)\) that could be derived, given a configuration option can either be selected or unselected. In a module, an import or context \(\Gamma\) 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 \(\Gamma \in C \rightarrow X \rightarrow P \rightarrow T\), where \(P\) is designed to show a projection relation. The method defined in a module \((\Delta)\) can only be compiled if the configuration is available as represented as \(C \rightarrow P \rightarrow X\). Furthermore, this project will be further projected to a set of expressions under certain types as \(T \times E\).

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 \(\Gamma\) and functions defined \(\Delta\). 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 \(\phi_{DUC\times TC}(m)\), 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 constrains 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:

\[
\phi_{DUC\times TC}(m),
\]

which will give a boolean value to show where a module \(m\) is well-type. 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.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e \in E)</td>
<td>expressions</td>
</tr>
<tr>
<td>(i \in T)</td>
<td>types</td>
</tr>
<tr>
<td>(x \in X)</td>
<td>function names</td>
</tr>
<tr>
<td>(o \in O)</td>
<td>configuration options</td>
</tr>
<tr>
<td>(c \in C = 2^O)</td>
<td>configurations</td>
</tr>
<tr>
<td>(v \in V = 2^c)</td>
<td>variability model</td>
</tr>
<tr>
<td>(\Gamma \in C \rightarrow X \rightarrow P \rightarrow T)</td>
<td>import function signature</td>
</tr>
<tr>
<td>(\Delta \in C \rightarrow X \rightarrow P \rightarrow T \times E)</td>
<td>function definition</td>
</tr>
<tr>
<td>(m = (v, i, j, \Gamma, \Delta) \in M; i, j \subseteq O)</td>
<td>module (^1)</td>
</tr>
</tbody>
</table>

\(^1\): configuration options imported, \(j\): configuration options defined
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 manner. 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 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 LF and discard the regular language RC, which ensures calling context sensitivity. LF gives a graphical representation G of a Java program and there are four canonical statements that could define edges in between:

- Allocation x = new 0: edge o \( \xrightarrow{\text{new}} \) x ∈ G
- Assignment x = y: edge y \( \xrightarrow{\text{assign}} \) x ∈ G
- Field write x.f = y: edge y \( \xrightarrow{\text{store}} \) x ∈ G
- Field read x = y.f: edge y \( \xrightarrow{\text{load}} \) x ∈ G

Within G, we use symbol flowsTo \( \rightarrow \) new (assign)* to indicate the new and assign edges. That is, 0 flowsTo v in G represents 0 within the points-to set of v. Moreover, the inverse symbol flowsTo is used to trace points-to for field access. Literally, if there is an flowsTo edge from 0 to v, there must be a flowsTo relation should from v to 0. 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 toStmtcondUn;
4 while TRUE do
5    for statement s in StmtcondE do
6        if there is a c flowsTo or \( \text{ flowsTo } \) 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 flowTo or flowTo relation to the statements in StmntcondE until the set does not change (line 4-9). And we apply a similar process for StmntcondUn.

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: \( \neg \text{batch} \land \text{doSplash} \rightarrow \text{splash} \), which mean the option splash is dependence on both \( \neg \text{batch} \) and \( \text{doSplash} \) 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**.

![Fig. 2. The core idea of our VMS approach](image)

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.

Specifically, within a module \( m \), it contains \( v \) for variability model, \( i \) for import configuration option, \( j \) for export option, \( \Gamma \) for import function signatures and \( \Delta \) for functions defined in module. The expression \( e \in E \) in a module could be extracted from ASTNode Expression. And the option \( o \in O \) 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 \( t \in T \) could be obtained using type-resolving techniques provided by Eclipse with an input of the binding of the type. As for the function definition \( \Delta \), it can be extracted by visiting all MethodDeclaration node in the AST. The rest of components in module \( m \), including, \( i, j \in O, \Gamma \) and \( v \in V \), could be built based on these basic elements using our definition in Section II-A.

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 a 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.

\[
  w_{tmr}(m, f) = \frac{1}{2n} \left( \sum_{m_i \in f} \left( \frac{\text{csd}(\text{def}(m), \text{ref}(m_i)) + \text{csd}(\text{ref}(m), \text{def}(m_i))}{\|X\| \|Y\|} \right) \right),
\]

where \( \text{targets}(m) = \{m' | (m, m') \in R\} \) and \( \text{sources}(m) = \{m' | (m', m) \in R\} \). 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 is 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: \( \text{def}(X) \) and \( \text{ref}(X) \). Specifically, the vector \( \text{def}(X) \in \mathbb{R}^n \) defines all types within \( X \) with \( n \) represent the number of types in the subject program. \( \text{def}(X) = [d_1, ..., d_n] \) is defined as if type \( i \) is defined in \( X \), then \( d_i = 1 \), otherwise \( d_i = 0 \). On the contrast, \( \text{ref}(X) = [r_1, ..., r_n] \) shows the reference information. If a type \( i \) is referenced by \( X \), then the \( i \)th element \( r_i \) in \( \text{ref}(X) \) should be 1, otherwise it should be 0. Therefore, the type reference distance \( w_{tr} \) is defined as follows.

\[
  w_{tr}(m, f) = \frac{1}{2n} \left( \sum_{m_i \in f} \left( \frac{\text{csd}(\text{def}(m), \text{ref}(m_i)) + \text{csd}(\text{ref}(m), \text{def}(m_i))}{\|X\| \|Y\|} \right) \right),
\]

where \( n \) is the number of modules currently in feature \( f \) and \( \text{csd} \) is defined as cosine similarity between two vectors:

\[
  \text{csd}(X, Y) = \frac{X \cdot Y}{\|X\| \|Y\|}.
\]

The subtility of this approach is it uses the cross reference to check how a module \( m_i \) in feature \( f \) relies on \( m \) with \( \text{csd}(\text{def}(m), \text{ref}(m_i)) \) 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 \( w \)s obtained from a Dirichlet distribution with the shape parameter \( \beta \). 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:

\[
  w_{dt}(m, f) = \frac{1}{n} \sum_{m_i \in f} \theta_{m_i} \cdot \theta_m,
\]

where \( \theta_m \) 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 \( (w_{tmr}, w_{tr} \text{ and } w_{dt}) \), we define an overview distance value as \( w_s \). The overview distance \( w_s \) is defined by following Robillard’s approach, which uses operator \( x \circ y = x + y - x \cdot y \) to combine two values [8]. This operator yields a result by equally treating all arguments and return its overall result within the range \([0, 1]\). Therefore, we calculate the overview distance by:

\[
  w_s = w_{tmr} \circ w_{tr} \circ w_{dt}.
\]

In addition, we have to put the variability-aware constraints on \( w_s \). The value of \( w_s \) represent the overall similarity between a module and a feature; the more \( w_s \) 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 \( (\Gamma_x(c) \cup \Gamma_y(c)) \) under a configuration \( c \in C \), (2) combining variability modules \( (v_x \cup v_y) \) and (3) merging imported and self-defined configuration options in each module \( (i_x \cup i_y \text{ and } j_x \cup j_y) \). 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.
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 module \( v' = v_x \cap v_y \setminus \text{conflict}(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y) \), 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 the conflict from both modules import functions with same name but different types; (2) defining \( \text{sig} \) gives a mapping \( \text{sig} : (X \to E \times T) \to (X \to T) \). Therefore \( \text{sig}(\Delta) \) returns a mapping \( X \to T \), which is a \( \Gamma \) 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 \( \Delta'(c) = \Delta_x'(c) \cup \Delta_y'(c) \); and (4) merge the configuration-option imports and remove the import configuration option from one module, but already defined in another \( \Gamma' = \Gamma_x \setminus \text{conflict}(j_x \cup j_y) \). 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.

\[
\phi (m_x \bullet m_y) = \begin{cases} 
1 & \text{if } m_x \cap m_y = \emptyset \\
0 & \text{otherwise}
\end{cases}
\]

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

**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 \#ef 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 \( \text{varPDG} \) and for each class/interface create a module;
- Line 7 - 12: 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 \( Q \)

**Algorithm 2: VMS feature model constructing approach**

**Input:** \#op, \#ef, \( P \)

**Output:** \( fm \)

1. Build the \( \text{varPDG} \) for input program \( P \);
2. Create an empty module to class/interface mapping \( D \);
3. for class \( e \) in \( P \) do
   4. create a module \( m \) using \( e \);
   5. add \((m, e)\) to \( D \);
6. Create set \( \text{CommonM} \);
7. Create empty module \( \text{main} \) to \( Q \);
   8. while \( Q \) not empty do
      9. \( f_{\text{head}} \leftarrow Q\cdot\text{poll} \);
      10. if \( f_{\text{head}} \) not visited then
         11. Find all functions \( \text{needprocess} \), which not defined in the scope of a condition \( \text{cond} \) in \( \text{varPDG} \);
         12. All all modules contain \( \text{needprocess} \) to \( \text{CommonM} \);
   13. For each module \( \text{commonM}_m \) in \( \text{commonM} \) into a cluster \( \text{commonM}_m \);
14. Let optional modules \( \text{optionalM} \) be \( \text{optionalM} \leftarrow D\cdot\text{keyset}() \setminus \text{commonM} \);
15. while \( \text{commonM} \cdot \text{size} > \#ef \) do
   16. Find two cluster \( \text{commonM}_i \) and \( \text{commonM}_j \) with maximum \( w_\ast (\text{commonM}_i, \text{commonM}_j) \) and \( (\text{commonM}_i, \text{commonM}_j, OK) \);
   17. Update all reference information;
18. while \( \text{optionalM} \cdot \text{size} > \#ef \) do
   19. Find two cluster \( \text{optionalM}_i \) and \( \text{optionalM}_j \) with maximum \( w_\ast (\text{optionalM}_i, \text{optionalM}_j) \) and \( (\text{optionalM}_i, \text{optionalM}_j, OK) \);
   20. Update all reference information;
21. Create \( fm \) from cluster recovered;
22. return \( fm \);
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 \(\text{option}M\).

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\(^3\) system. The TypeChef\(^3\) 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 Lancaster, 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, SMS Transfer \(\Rightarrow\) 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/kaestne/LEADT

\(^4\)Prevayler: available at http://prevayler.org

\(^5\)ArgoUML: available at: http://argouml.tigris.org


\(^7\)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:

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

where $\text{MoJo}(A, B) = \min(\text{mno}(A, B), \text{mno}(B, A))$ and $\text{mno}(A, B)$ 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 $\text{mno}(A, B)$. Furthermore, the symbol $\forall A$ in the denominator represents a partition of $A$ and $\max(\text{mno}(\forall A, B))$ means the maximal distance from any partition $A$ to $B$.

### Table III

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Prevalyer</th>
<th>MobileMedia</th>
<th>ArgoUML</th>
<th>BerkeleyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDC</td>
<td>83.0</td>
<td>75.0</td>
<td>63.69</td>
<td>83.93</td>
</tr>
<tr>
<td>LIMBO</td>
<td>55.56</td>
<td>68.75</td>
<td>52.03</td>
<td>78.33</td>
</tr>
<tr>
<td>Bunch</td>
<td>74.07</td>
<td>68.42</td>
<td>63.92</td>
<td>88.77</td>
</tr>
<tr>
<td>ARC</td>
<td>59.25</td>
<td>73.43</td>
<td>53.09</td>
<td>84.95</td>
</tr>
<tr>
<td>VMS</td>
<td>83.33</td>
<td>71.87</td>
<td>79.78</td>
<td>81.62</td>
</tr>
<tr>
<td>W-UE</td>
<td>66.67</td>
<td>78.12</td>
<td>51.66</td>
<td>83.41</td>
</tr>
<tr>
<td>W-UENM</td>
<td>68.52</td>
<td>71.87</td>
<td>51.66</td>
<td>82.14</td>
</tr>
</tbody>
</table>

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:

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

3) **Cluster-to-cluster Coverage(c2c)**: c2c explores the component-level accuracy and is given as [34]:

$$
c2c(c_i, c_j) = \frac{|\text{entities}(c_i) \cap \text{entities}(c_j)|}{\max(|\text{entities}(c_i)|, |\text{entities}(c_j)|)} \times 100\%,
$$

where $c_i$ is a cluster generated by clustering techniques and $c_j$ is the cluster from the ground-truth. The $\text{entities}(c)$ shows all candidates in the cluster $c$. The architecture coverage $c2c_{cvg}$ is a metric that extend the clusters overlap as:

$$
c2c_{cvg}(c_1, c_2) = \frac{\text{simC}(A_1, A_2)}{\max(|\text{entities}(A_1)|, |\text{entities}(A_2)|)} \times 100\%,$$

where $\text{simC}(A_1, A_2) = \{c_1 | c_1 \in A_1, \exists c_2 \in A_2 \land (c2c(c_1, c_2) > \text{th}_{cvg})\}$.

### Table IV

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Prevalyer</th>
<th>MobileMedia</th>
<th>ArgoUML</th>
<th>BerkeleyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDC</td>
<td>71.78</td>
<td>30.31</td>
<td>51.46</td>
<td>17.97</td>
</tr>
<tr>
<td>LIMBO</td>
<td>47.19</td>
<td>63.68</td>
<td>47.67</td>
<td>55.57</td>
</tr>
<tr>
<td>Bunch</td>
<td>45.07</td>
<td>56.75</td>
<td>49.06</td>
<td>49.08</td>
</tr>
<tr>
<td>ARC</td>
<td>49.23</td>
<td>67.39</td>
<td>49.76</td>
<td>63.48</td>
</tr>
<tr>
<td>VMS</td>
<td>52.16</td>
<td>67.70</td>
<td>52.87</td>
<td>62.92</td>
</tr>
<tr>
<td>W-UE</td>
<td>50.00</td>
<td>73.56</td>
<td>49.90</td>
<td>62.01</td>
</tr>
<tr>
<td>W-UENM</td>
<td>50.00</td>
<td>73.56</td>
<td>49.90</td>
<td>61.64</td>
</tr>
</tbody>
</table>

The results shown in Table III, Tab. IV, and associate violin plots, including Fig.3 and Fig.5, indicate that at the system level, 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.

8Metrics 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%))

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

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 \( w_s \) 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 architecture-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’s 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**


