HAL
open science

Toward Recovering Component-based Software Product Line Architecture from Object-Oriented Product Variants

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai

To cite this version:

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai. Toward Recovering Component-based Software Product Line Architecture from Object-Oriented Product Variants. SEKE: Software Engineering and Knowledge Engineering, Jul 2016, San Francisco, United States. pp.1-7, 10.18293/SEKE2016-066 . lirmm-01376027

HAL Id: lirmm-01376027

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01376027v1

Submitted on 9 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Toward Recovering Component-based Software Product Line Architecture from Object-Oriented Product Variants

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai
UMR CNRS 5506, LIRMM, University of Montpellier 2 for Sciences and Technology, France
E-mail: {Eyalsalman, Seriai}@lirmm.fr

Abstract

Usually, companies meet different customer needs in a particular domain by developing variants of a software product. This is often performed by ad-hoc copying and modifying of various existing variants to fit purposes of new one. As the number of product variants grows, such an ad-hoc development causes severe problems to maintain these variants. Software Product Line Engineering (SPLE) can be helpful here by supporting a large-scale reuse systematically. SPL architecture (SPLA) is a key asset as it is used to derive architecture for each product in SPL. Unfortunately, developing SPLA from scratch is a costly task. In this paper, we propose an approach to contribute for recovering SPLA from existing product variants. This contribution is two-fold. Firstly, identifying common features and variation points of features of a given collection of product variants. Secondly, exploiting commonality and variability in terms of features to identify mandatory components and variation points of components as an important step in this recovering process. To evaluate the proposed approach, we applied it to two case studies. The experimental results bring evidence the effectiveness of our approach.

Keywords*: product variants, software product line architecture, component, variability, feature, recovering.*

1 Introduction

It is common for companies developing variants of a software product to accommodate different customer needs. These variants provide common features and differ from one another by providing unique feature combinations. A feature is "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [1]. Companies develop separate product variants where a new product is built by ad-hoc copying and modifying of various existing variants to fit purposes of new one. Separately maintaining these variants causes challenges. Changes in the source code corresponding to common features (e.g., for bug fixing) must be repeated across product variants. Therefore, companies need to change their software development strategy by a transition to a systemic reuse approach, such as Software Product Line Engineering (SPLE) to avoid such maintenance problems.

Reuse is a main characteristic of SPLE. It builds core assets consisting of all reusable software artifacts. SPL architecture (SPLA) is a key asset [2]. It is a core architecture that captures high level design decisions for SPL products, including the variation points and variants. These decisions concern in the organization of components and general rules that these components have to obey. Commonality and variability in SPLA (i.e., mandatory components and variation points (VPs) of components) globally originate from commonality and variability of customers' requirements/features which are represented by feature model [3].

Developing SPLA from scratch is a costly task because it should encompass the components realizing all the mandatory and varying features in a particular domain. Therefore, it is useful to exploit product variants for reducing the development cost. Recovering SPLA from software product variants is not considered in the literature. Existing works support recovering software architecture from single existing software system [4]. In this paper, we present an approach to contribute for recovering SPLA from existing product variants. Our contribution is two-fold. Firstly, identifying common features and VPs of features as this organization of features represents the main source of commonality and variability in SPLA. Secondly, exploiting such identification to identify mandatory components and VPs components for SPLA. We adapt our previous component extraction approach (*ROMANTIC*) approach to extract components [5].

The remainder of the paper is organized as follows: we give a necessary background in section 2. Section 3 detail the proposed approach steps. Next, section 4 shows experimental results and analysis. Sections 5 and 6 discuss the related work and conclude the paper, respectively.

2 Variation Points in Feature Model and Software Product Line Architecture

Development of SPLs mainly relies on exploiting commonality and managing the variability between SPL's products to meet all customer requirements [3]. Feature Model (FM) is a well-known artifact to represent this commonality and variability in terms of features. Commonality refers to feature(s) that are a mandatory part of each product. FM offers three types of feature groups: *XOR-Group* (alternative), *OR-Group* and *AND-Group* (see Figure 1). Also, we consider all optional features fallen down from FM root as a single feature group called *OP-Group*. Each feature group represent a VP at the feature level. Such a VP is reflected in SPLA as a VP of components. An example to clarify the concept of VP in both SPLA and FM is shown in Figure 2. This figure shows all available alternatives for a customer to choose a phone that only supports *color*, *high resolution* or *basic* screen. It shows that each screen option is realized by a combination of two components and selection the appropriate combination is based on customer choose. Therefore, an explicit link between features and components is needed to bind variability in SPLA (VPs) according to customers' needs.

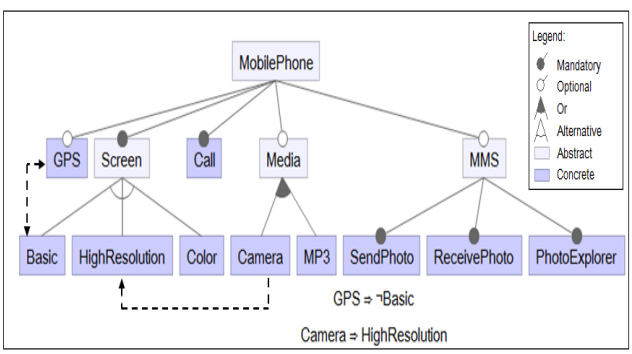

Figure 1: FM of MobilePhone-SPL [6].

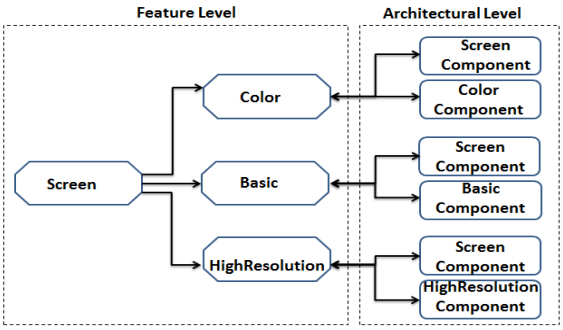

Figure 2: An Example of a Variation Point at Feature and Architecture Levels.

In our previous work [5], we proposed ROMANTIC approach to automatically recover a component-based architecture from the source code of a single existing object-oriented software. Component is a cluster of classes collaborating to provide a function of a software system. In this paper, we reuse this approach to extract components from the implementation of each feature group (VP) and common features.

3 The Proposed Approach

As SPLA encompasses commonality and variability of customers' requirements/features, we should first identify mandatory features (commonality) and VPs of features (variability) across product variants. Then, we need to exploit this commonality and variability in SPLA as mandatory components and VPs of components respectively. In our approach, components are extracted from the implementation of each group of features (i.e., mandatory features group and VPs).

Our proposed approach is organized into two phases. In the first phase, we identify mandatory features and VPs of features. This phase takes as input product configurations and feature descriptions. In the second phase, we identify mandatory components and VPs of components. This phase takes as input mandatory features, VPs of features and all feature implementations (as source code classes). This phase starts by extracting components from classes of each VP of features and from mandatory features group. Next, each feature is linked to its corresponding component(s) for binding commonality and variability at the architectural level in order to identify mandatory components and VP of components.

3.1 Identifying Mandatory Features and Variation Points of Features

We analyze the behavior of members of each VP across product configurations of a given collection of product variants. A product configuration is a combination of features supported by a product. This behavior represents a pattern of VPs. We propose algorithms to detect VP pattern by considering the definition of each type of VPs.

3.1.1 Basic Definitions

To explain our proposed algorithms, we use a FM inspired from the mobile phone industry (see Figure 1) to generate valid configurations [6]. We treat the generated configurations as product variants configurations. Table 1 shows all possible valid configurations that can be generated from FM in Figure 1. The check symbol (✓) refers to the features

provided by each configuration. Before presenting our algorithms, we start by defining the key concepts which are shared among the proposed algorithms.

Definition 1 (Feature List)**.** *Feature list (FL) is a list of all unique features in all product configurations. All features provided by FM shown in Figure 1 represent an example of FL.*

Definition 2 (Feature Set)**.** *Feature set is a tuple* $FS = [sef, \overline{sef}]$ *where* sef *and* $\overline{sef}$ *are respectively the set of selected and not-selected features of a product [7]. Thus* $sef \cap \overline{sef} = \Phi$ *and* $sef \cup \overline{sef} = FL$*. The* $P.sef$ *and* $P.\overline{sef}$ *respectively refer to the set of selected and not-selected features of a product P.*

An example of a FS is product $P_{18} = [\{$*Call, SendPhoto, ReceivePhoto, PhotoExplorer, HighResolution, GPS*$\}, \{$*Basic, Color, Camera, MP3*$\}]$ in Table 1.

Definition 3 (Feature Set Table)**.** *Feature set table (FST) is a collection of* FSs*. Each row in this table represents a product so that for every product* P_i *we have* $P_i.sef \cup P_i.\overline{sef} = FL$*. Table 1 is an example of FST.*

3.1.2 Identifying Mandatory Features

Mandatory features can be identified simply by comparing feature names of given product configurations (*FST*) to find features that are part of each configurations. Then, we prune FST and FL by excluding mandatory features from their contents.

3.1.3 Identifying AND Variation Points of Features

The behavior of an AND-Group of features across product variants seems like an atomic set that its members always appear or disappear together.

Based on this regular behavior, we start by intersecting pair-wisely all selected feature sides ($P_i.sef$) for all products in *FST*. This intersection aims at finding candidate sets that their members (features) may appear together. Of course, not each set represents AND-Group. Therefore, we pair-wisely check the members of each set against the semantic of the AND-Group. Any pair that does not respect this semantic is rejected. Consequently, each resulted set is either an AND-Group only consisting of two members or a pair of features that comply to a *require* constraint like *HighResolution* and *Camera* features in Figure 1. Therefore, we use feature descriptions to filter out pairs that comply to a *require* constraint by conducting textual matching between terms of feature descriptions. The idea behind using feature descriptions is that features that belong to the same group should have common terms in their descriptions. As a result, the remaining pairs represent only AND-Group consisting of two features. Then, we merge together pairs that have transitive relations to form AND-Groups of three or more members. Finally, we prune *FST* and *FL* by excluding AND-Groups members.

Table 1: Product Configurations of Mobile Phone SPL

Product	Ca	S	R	P	B	H	Co	Cam	M	G
P1	✓				✓					
P2	✓					✓				
P3	✓						✓			
P4	✓					✓				✓
P5	✓						✓			✓
P6	✓					✓		✓		
P7	✓				✓				✓	
P8	✓					✓			✓	
P9	✓						✓		✓	
P10	✓					✓		✓	✓	
P11	✓					✓		✓		✓
P12	✓					✓			✓	✓
P13	✓						✓		✓	✓
P14	✓					✓		✓	✓	✓
P15	✓	✓	✓	✓	✓					
P16	✓	✓	✓	✓		✓				
P17	✓	✓	✓	✓			✓			
P18	✓	✓	✓	✓		✓				✓
P19	✓	✓	✓	✓			✓			✓
P20	✓	✓	✓	✓		✓		✓		
P21	✓	✓	✓	✓	✓				✓	
P22	✓	✓	✓	✓		✓			✓	
P23	✓	✓	✓	✓			✓		✓	
P24	✓	✓	✓	✓		✓		✓	✓	
P25	✓	✓	✓	✓		✓		✓		✓
P26	✓	✓	✓	✓		✓			✓	✓
P27	✓	✓	✓	✓			✓		✓	✓
P28	✓	✓	✓	✓		✓		✓	✓	✓

Note: we use as column labels the shortest distinguishable prefix of the feature names (e.g. *Co* for *Color* feature shown in Figure 1)

3.1.4 Identifying XOR Variation Points of Features

The behavior a XOR-Group members across product configurations imposes that the existence only one member in $P_i.sef$ while the remaining members in $P_i.\overline{sef}$. Based on this behavior, we propose Algorithm 1 to identify XOR-Groups of features. The main data structures are used through the algorithm are *Multiset* and *HashMap* (line 1). In lines (2-6), we assume that each feature (*F*) in *FL* is a member of a XOR-Group. Therefore, if *F* is provided by many products in FST, we obtain many sets corresponding to *F*. Of course, not all elements of these sets have exclusive relations with F. Therefore, we intersect all these sets to filter out irrelevant elements (features) as much as possible. After the intersection, F corresponds to a unique set. Then, F and its corresponding set is kept as an entry in a HashMap called *ExRe*. Each entry represents *excluded-relation* that takes the following formate *[F* $\Leftrightarrow$ *set of features]*. *F* represents the left-hand side (LHS) while its corresponding set represents the right-hand side (RHS). Figure 3 shows *excluded-relations* obtained from our illustrative example.

The elements of RHS of an entry in ExRe are a combination of features so that this combination may only consist of members of the same XOR-Group's or it may include members of other XOR-Groups. Therefore, lines (7-14) check

Algorithm 1: Identifying XOR-Group Variation Points

Input: FST, FL, Feature Descriptions
Output: XGF (**X**OR-**G**roups of **F**etures)

1 XGF ← ϕ, //multiset ExRe ← ϕ //HashMap
2 **foreach** $F \in FL$ **do**
3 **foreach** *i from 1 to* $|FST|$ **do**
4 **if** $F \in P_i.sef$ **then**
5 intersect ← intersect ∩ P_i.sef
6 ExRe.put(F, intersect) //ExRe.put(key, value)
7 **foreach** *entry En* ∈ *ExRe* **do**
8 Set RHS ← En.getValue(), count ← 0
9 Set CurExRe ← RHS, add(CurExRe, En.getKey())
10 **if** !(*CurExRe* ∈ *XGF*) **then**
11 **foreach** *feature f* ∈ *RHS* **do**
12 Set temp1 ← CurExRe, remove (temp1, f)
13 **if** *temp1* ⊆ *ExRe.getValue(f)* **then**
14 count++
15 **if** *count* = $|RHS|$ **then**
16 add(XGF, CurExRe)
17 XGF ← ApplyFeaDes(XGF)
18 Purne(FST, FL, XGF)
19 return XGF

each entry in *ExRe* against the definition of XOR-Group. Considering that an entry (`En`) in `ExRe` is a XOR-Group, this means that each feature (`f`) in the RHS of *En* must appear as a LHS of another *excluded-relation* (*Ex1*) and RHS of *Ex1* must contain all `En's` features except *f*. By applying these lines, entries 2, 3 and 4 in Figure 3 are identified as candidate XOR-Groups and put in XGF while others are rejected (lines 15-16). XGF may contain groups only consist of two members (e.g., entry 3 in Figure 3). Such groups are not necessary to represent XOR-Groups. They may be pairs of features that comply to an *exclude-constraint* such as *GPS* and *Basic* features in Figure 1. Therefore in line 17, we use feature descriptions (*ApplyFeaDes()*) to identify and then remove such groups from XGF. Additionally, XGF may contain groups have the semantic of a XOR-Group but in fact their members can not be aggregated as a group. Such a situation occurs due to cross tree constraints (i.e., require and exclude constraints). For example, entry 2 in Figure 3 is identified as XOR-Group in spite of *Camera* feature in this entry belongs to OR-Group. This happened due to the cross tree constraint between *Camera* and *HighResolution*, which create an alternative relation between *Camera* and both *Basic* and *Color* (see Figure 1). In this situation, we use also feature descriptions to identify such groups. In line 18, we prune *FST* and *FL* by excluding members of XOR-Groups.

1-[Basic ↔ Camera, Color, GPS, HighResolution]
2-[Camera ↔ Basic, Color]
3-[GPS ↔ Basic]
4-[HighResolution ↔ Basic, Color]
5-[Color ↔ Basic, Camera, HighResolution]

Figure 3: All Excluded-Relations of FM in Figure 1.

3.1.5 Identifying OR and OP Variation Points of Features

Although the remaining features in FST are only OR-Groups and OP-Groups, the identification of their members is a challenge because the behavior of members of these groups is arbitrarily. For the identification of OP-Groups, we completely rely on feature descriptions. We consider features that have common keywords in their descriptions belong to the same OR-Group. In this way, we can identify OR-Groups. After identifying OR-Groups, the remaining features represent the members of a single OP-Group.

3.2 Identifying Mandatory Components and Variation Points of Components

3.2.1 Component Extraction

In our approach, a component is extracted based on ROMANTIC approach as a cluster of classes. ROMANTIC is applied to source code classes implementing mandatory features, AND-Group, XOR-Group, OR-Group and OP-Group. Such an application respects the components organization in SPLA (i.e., mandatory components and VPs of components). Components that are extracted from the implementation of mandatory features constitute the mandatory components while components extracted from each feature group constitute members of a VP in SPLA.

The implementations of feature group may have shared source code classes. Such classes are not specific to a certain feature group and they implement features having cross cutting behavior across all other features. Therefore, we determine such classes and then we apply ROMANTIC to these classes as a group. The extracted components represent a group of component called *Shared-Com*. They are not specific for a certain VP in SPLA. The selection of these components for products development depends on the selection of their associated features.

3.2.2 Recovering Feature-to-Component Traceability Links

From the previous step, we notice that components of a VP are not necessary having the semantic of that VP. For example, consider that *F1* and *F2* are two features which belong

to XOR-Group, and components extracted from the source code implementing this group are [com1, com2, com3 and com4]. Also, assume that the first three components implement `F1` while `com4` implements `F2`. In this case the relation among the first three components is not exclusive although they belong to exclusive VP. This is because the mapping between components and features is many-to-many [3]. This means that a feature's implementation may be scattered over more than one component inside a VP and also a component may implement more than one feature. This mapping should be considered during the creation of VPs through establishing explicit links between features and their corresponding components in SPLA. These links are useful for binding variability in SPLA. Such links determine a combination of components inside each VP so that each combination represents a variant of that VP. Determining variants of each VP in SPLA is important to specify the constraints among components.

Such traceability links between features and components can be established by exploiting the transitive relation between features and components through source code classes. A feature is implemented by classes and a component is composed of classes. Therefore, classes are a shared element between features and components, which allow linking them together. After such linking, it is normal to have shared components between all features belonging to the same VP due to the many-to-many mapping between features and components. Therefore, these components are not specific to a certain feature but they are related to the parent of those features and form a parent of VP of components in SPLA.

4 Experimental Results and Evaluation

We organize our evaluation into two parts. In the first part, we evaluate the algorithms used to identify mandatory features and VPs of features. In the second part, we evaluate the identification of mandatory components and VPs of components.

4.1 Case Studies

To evaluate our approach, we apply it to two case studies: *ArgoUML-SPL* and *MobileMedia*. ArgoUML-SPL is the SPL for the UML modeling tool *ArgoUML*. It is open source JAVA application and provides nine features. These features are organized as a mandatory feature, an OR-Group and an OP-Group. It supports two features (*Cognitive Support* and *Logging*) that have crosscutting behavior through all other features [8]. We obtain the description of each feature of ArgoUML-SPL through its official website and manual instructions[1]. Due to space limitation we can not be able to present ArgoUML-SPL's FM. MobileMedia is a JAVA open source which manipulates multimedia on mobile devices. It was implemented in *8* subsequent releases. Each release represents a variant corresponds to an evolutionary step of the system development. We only consider releases (1-3 and 5-6) due to the nature of the evolution, as features in excluded releases do not have the same implementation in these releases. The description of MobileMedia's features are obtained by official website of MobileMedia[2], descriptions of its use cases and analyzing source code comments.

4.2 Validating the Identification of Mandatory Features and VPs of Features

As a base for evaluation, we match each VP identified by our approach with its corresponding VP in the focused FM. This matching is measured by using two metrics inspired from information retrieval field, namely *Precision* and *Recall*. *Precision* measures the accuracy of identifying members of a VP according to the relevant members of that VP. *Recall* measures to what degree the members of identified VP covers the relevant members of that VP. The relevant members of each VP are determined from the FM. All measures have values within [0,1]. Our proposed algorithms aims to achieve high precision and recall. We propose the equations 1 and 2 to adapt *Precision* and *Recall* in our context. *IM_VP* and *RM_VP* in these equations represent respectively Identified and Actual members of VP_i.

We randomly generate two sets from ArgoUML-SPL's FM and three sets from MobileMedia using FeatureIDE tool. Each generated set has different size and it also covers all features shown in its corresponding FM. The *set1* in all case studies represents all possible configurations can be generated from its corresponding FM.

$$Precision(VP_i) = \frac{|IM_VP_i \bigcap RM_VP_i|}{|IM_VP_i|} \times 100\% \quad (1)$$

$$Recall(VP_i) = \frac{|IM_VP_i \bigcap RM_VP_i|}{|RM_VP_i|} \times 100\% \quad (2)$$

Table 2 shows obtained results by applying our algorithms to product configurations generated from FMs of case studies considered. In this table, we present Precision and Recall for each identified VP and average Precision and Recall for all identified VPs corresponding in each set of configurations. Highlighted rows refer to VPs that identified by our algorithms but they actually are not present in FMs of cases studies considered (false-positive VPs).

In MobileMedia case study, the proposed algorithms give 100% *Precision* and 100% *Recall* for each VP in both

[1] http://argouml-spl.stage.tigris.org/

[2] http://www.ic.unicamp.br/ tizzei/mobilemedia/

Table 2: Precision and Recall of Identified VPs of Features for Both ArgoUML-SPL and MobileMedia.

ArgoUML-SPL				
Set1: No. Configurations = 256 (All possible configurations)				
	Precision	Recall	Average Precision	Average Recall
Mandatory	100%	100%	58%	67%
OR-Group VP1	67%	100%		
OP-Group VP1	0.0%	0.0%		
Set2: No. Configurations = 7				
Mandatory	100%	100%	58%	67%
OR-Group VP1	67%	100%		
OP-Group VP1	0.0%	0.0%		
MobileMedia				
Set1: No. Configurations = 16 (All possible configurations)				
Mandatory	100%	100%	100%	100%
OR-Group VP1	100%	100%		
OP-Group VP1	100%	100%		
Set2: No. Configurations = 8				
Mandatory	100%	100%	25%	25%
OR-Group VP1	0.0%	0.0%		
OP-Group VP1	0.0%	0.0%		
XOR-Group VP1	0.0%	0.0%		
XOR-Group VP2	0.0%	0.0%		
Set3: No. Configurations = 5				
Mandatory	100%	100%	100%	100%
OR-Group VP1	100%	100%		
OP-Group VP1	100%	100%		

set1 (containing all possible configurations) and *set3* (containing only 5 configurations). This is because these configurations have a high diversity of feature combinations to detect the behavior of members of each VP. In set 2, although the number of configurations is half of all possible configurations, the proposed algorithms fails to identify two VPs (*OR-Group* VP1, *OP-Group* VP1) and return two false positive VPs (*XOR-Group* VP1, *XOR-Group* VP2).

In ArgoUML-SPL, the identified VPs from *set1* and *set2* have the same *Precision* and *Recall* values in spite of *set1* represents all possible configurations while *set2* represents very small number of configurations. This is due to the fact that the FM of ArgoUML-SPL just offers two types of VPs (*OR-Group* VP1, *OP-Group* VP1) which this means that we only rely on feature descriptions to identify these VPs and do not pay attention to the number of available configurations. We also notice that the algorithms failed to identify *CognitiveSupport* and *Logging* (their label is *OP-Group* VP1) as VP of type *OP-Group* but they are identified as *OR-Group*. This is because these features share keywords with all other features as they have a crosscutting behavior in all ArgoUML-SPL features. Therefore, (*CognitiveSupport* and *Logging*) are identified as members of *OR-Group* VP1. This leads to degrade *Precision* and *Recall* values of *OP-Group* VP1 and *OR-Group* VP1 as shown in Table 2.

4.3 Validating the Identification of VPs of Components

For space limitation, we present only the result of applying our approach to ArgoUML-SPL case study. We generate 7 products corresponding to the second set of configurations in ArgoUMl-SPL (see Table 2).

Table 3: Common Components and Variation Points at Architectural Level for ArgoUML-SPL.

At Architecture Level: OR-VP1	
At Feature Level: OR-Group VP1 (State, Activity, UseCase, Collaboration, Deployment, Sequence)	
Parent Components	**Member Components**
Fig Prop Diagram Action Mode List State	Action Fig Button New Prop State Selection
Fig Diagram State Sequence Model Activity Renderer	UML Model List Fig Prop Case Use
UML Fig Diagram Init Model Action List	Action Fig Mode Iterator Message State Attributes
Go Fig To State Diagram Collaboration Machine	
At Architecture Level: OP-VP1	
At Feature Level: OP-Group VP1 (CognitiveSupport, Logging)	
Parent Components	**Member Components**
No Parent Components	Cr Wiz Child Many Conflict No Gen
	Cr To Name Do By Missing Class
	To Go Cr Init Wiz
	Cr Resolved Abstract Transitions Name
	Wiz Default Step To Renderer Tree Prop
	Cr Goals Dialog Add Param Type To
	Node Decision Knowledge Goal Priority Type
	Cr Go Wiz No Name Operation Invalid
	List Action To Object Tab Do
	Table Cr Checklist To Abstract UML Model
At Architecture Level: Mandatory Components	
At Feature level: Mandatory Features: Class	
Parent Components	**Member Components**
No Parent Components	Fig Style List Panel Class diagram Model
	Package Character Port Rect Fig
	Action Show Hide Visibility Stereotype
	Classdiagram Fig Subsystem Association Edge
	Classdiagram Fig Event Edge Object
	Selection Fig Class Node List Stereotype
At Architecture Level: Shared-Com	
Cr Without Instance Classifier	
Cr Without Class Component	
Cr Without Instance Node Comp	
Cr Without Instance Classifier Node Collection Iterator	
Cr Without Node Component	
Cr Without Instance Classifier Component	
Cr Interface Without Component	

Table 3 shows the identified mandatory components and VPs of components by applying our approach to the 7 products of ArgoUML-SPL code base. *Parent Components* column presents components that form the parent of a VP while *Member Components* column presents components that form members of that VP. In this table, we show for each VP at the architecture level its corresponding VP at the feature level. The name of the identified VPs of components in this table as a follows: *OR-VP1* and *OP-VP1*. For components of *OR-VP1*, we notice that all names of parent components share the term "Diagram". This means that they are not specific to certain feature (UML Diagram) but they may support all UML diagrams. By referring to ArgoUML-SPL's FM, we can find out that the group of features corresponding to *OR-VP1* is under title "Diagrams". Consequently, our approach can distinguish between parent components and member components. For components of *OP-VP1* which is corresponded to *CognitiveSupport* and *Logging* features, our approach identifies no parent components for this VP. This is because its components are only extracted from the implementation of (*CognitiveSupport*) while the *Logging* feature is implemented by external library (Log4J) [8]. From the names of these components, we can notice that they specific to *CognitiveSupport* because their names contain "Cr" term which refers to *Critics* supported by *CognitiveSupport* feature. For mandatory

components which are extracted from the implementation of mandatory feature (Class diagram), the names of these components show that these components implement this feature as their names contain the term "Classdiagram". Of course, there is no parent components for mandatory components because there is only one feature. For components of *Shared-Com*, all components of this group related to only *CognitiveSupport* as their names include "Cr" term. This is expected because this feature has across cutting behavior through all other features. This means the implementation of this feature is shared among the implementation of other feature groups, which that represents the semantic of *Shared-Com* group. Consequently, our approach can extract and determine components that are shared between VPs of components (*Shared-Com*).

5 Related Work

The existing works support only forward engineering way for building SPLA. In [9], Sochos et al. propose to create SPLA based on FM. A strong mapping between features and components are established based on four transformations on the initial FM leading to SPLA. According to their approach components are developed from scratch to implement the transformed features. In [10], Trinidad et al. propose to automatically build a component model from a FM for developing dynamic SPL. They create for each feature a component and relations among features become relations among components. In [11], Zhang et al. propose to map feature to architectural components for building SPLA. In their approach, features are classified according to the variability type into *mandatory* and *optional*. In their approach, a component is created for each feature. The components of crosscutting features are implemented by object-oriented techniques while the components of non-crosscutting features are implemented by aspect-oriented techniques.

6 Conclusions

In this paper, we have proposed an approach for recovering SPLA from software product variants. Our approach focused on identifying mandatory components and variation points of components as an important step toward recovering SPLA. We analyzed commonality and variability across product variants in terms of features, as they represent the main source of commonality and variability in SPLA. In our experimental evaluation using two case studies, we showed that if we have small number of configurations with high diversity and precise feature descriptions, our approach achieves high precision and recall of identified variation points of features, and hence this leads to identify relevant variation point of components.

References

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," November 1990.

[2] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[3] K. Pohl, G. Bckle, and F. J. van der Linden, "Software product line engineering: Foundations, principles and techniques." Springer Publishing Company, Incorporated, 2010.

[4] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *J. Syst. Softw.*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[5] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of component-based architecture from object-oriented systems," in *WICSA*, 2008, pp. 285–288.

[6] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, Sep. 2010.

[7] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Reverse engineering feature models from programs' feature sets," ser. WCRE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 308–312.

[8] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 191–200.

[9] P. Sochos, M. Riebisch, and I. Philippow, "The feature-architecture mapping (FArM) method for feature-oriented development of software product lines," ser. ECBS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 308–318.

[10] P. Trinidad, A. R. Corts, J. Pena, and D. Benavides, "Mapping feature models onto component models to build dynamic software product lines." in *SPLC (2)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007, pp. 51–56.

[11] J. Zhang, X. Cai, and G. Liu, "Mapping features to architectural components in aspect-oriented software product lines," ser. CSSE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 94–97.