Understanding Feature Evolution in a Family of Product Variants

Yinxing Xue, Zhenchang Xing and Stan Jarzabek
School of Computing
National University of Singapore
{yinxing,xingzc,stan}@comp.nus.edu.sg

Abstract

Existing software product variants, developed by ad hoc reuse such as copy-paste-modify, are often a starting point for building Software Product Line (SPL). Understanding of how features evolved in product variants is a prerequisite to transition from ad hoc to systematic SPL reuse. We propose a method that assists analysts in detecting changes to product features during evolution. We first entail that features and their inter-dependencies for each product variant are documented as product feature model. We then apply model differencing algorithm to identify evolutionary changes that occurred to features of different product variants. We evaluate the effectiveness of our approach on a family of medium-size financial systems. We also investigate the scalability of our approach with synthetic data. The evaluation demonstrates that our approach yields good results and scales to large systems. Our approach enables the subsequent variability analysis and consolidation of product variants in the task of reengineering product variants into SPL.

1 Introduction

Product variants often evolve from an initial product developed for and successfully used by the first customer. Figure 1 presents five product variants of the Wingsoft Financial Management System (WFMS) [39], spawned from WFMS^{Fudan} developed for Fudan University. The successful deployment of the initial product has attracted new customers, such as Zhejiang University and Shanghai University. WFMSes have now been used in over 100 universities in China.

Initially, WFMS developers copy-paste-modify existing product variants when building a new product variant. For example, WFMS^{Chongqing} was built by adapting WFMS^{Fudan} and WFMS^{Zhejiang}. Such ad hoc reuse becomes problematic as the number of features and the number of product variants grows [30]. Not only do we have to maintain each product variant separately from others, but it also becomes difficult to find and adapt features for reuse in new products [39].

As these problems accumulate, it is worth reengineering product variants into a Software Product Line (SPL) for systematic reuse [9]. Extractive reengineering [22][23] into SPL is a low cost approach in which the initial reusable core assets include only features already implemented in existing product variants. The first step in extractive approach is therefore to understand common and variant features in existing product variants.

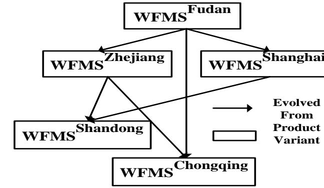

Figure 1. The variants of WFMS product family

In this paper, a feature represents a “distinctive use-visible aspect of a software system” [19][20]. Features have unique names and informal descriptions. Features can be organized in a hierarchy in which general composite features (e.g., *FeePayment*) are decomposed into more specific sub-features (e.g., *OnlinePayment* and *WSPayment*) [39].

At first glance, it seems to be an easy task to identify common and variant features in product variants. However, the problem is non-trivial for a big product family with many features that has been evolved for long time. During evolution, feature names and descriptions as well as the dependencies and relationships between features might have been changed. New features might have been added and existing features might have been deleted. The features might have also been split or merged.

For example, feature *Zhifu* in WFMS^{Fudan} was later renamed to *Payment* in subsequent product variants. *Zhifu* is a word in Chinese Pinyin, which means payment. WFMS^{Fudan} supports only the *Direct* authentication mode, while two more authentication modes *IDCard* and *SSO* (*Single Sign-On* mode) were subsequently introduced in WFMS^{Zhejiang} and WFMS^{Chongqing}, respectively. Feature *WSPayment* used to be a sub-feature of *OnlinePayment* in WFMS^{Fudan}, but it was made a sibling feature of *OnlinePayment*, since *WSPayment* supports the payment using mobile network in subsequent variants. Feature *InitBasicInfo* in WFMS^{Fudan} was split into two features, *InitBasicInfo* and *InitCsDbUser*, to support a new way of system initialization in subsequent variants.

To understand the feature evolution in WFMS, existing work on domain analysis and requirement

engineering has been focused on developing modeling techniques, such as goal model [10] and feature model [20] to capture and analyze requirements, monitoring and tracing requirement changes [18][40], and reasoning about the consistency and configurability of requirement models [4][34]. However, there is a lack of automatic tools that can produce an accurate report of feature evolution in a family of product variants.

In this paper, we present a method to detect changes that occurred to product features in a family of product variants. The primary input to our method is a set of Product Feature Models (PFMs). A PFM captures all the features and their dependencies in a product variant. The PFMs can be provided by system experts of the subject product variants. They may also be reverse-engineered from the implementations of product variants using feature location methods [2] [13][21][31].

We then adapt *GenericDiff* [38], a general framework for model comparison, to compare pair-wisely these PFMs based on both lexical and structural (i.e., dependencies and relationships) similarities of features. The differencing report between two PFMs is analyzed to determine: 1) the "same" features but with different names in two product variants; 2) the "same" features but belonging to different composite features in two product variants; 3) the splitting and merging of features; and 4) the features that are unique in two product variants.

We evaluate the effectiveness of our method by applying it to the product family of Wingsoft Financial Management System (WFMS). We also design a controlled experiment to evaluate the scalability of our method with a large volume of synthesized PFMs. Our evaluation demonstrates that our approach yields good results for identifying feature changes in product variants and scales to large systems. Understanding feature evolution in product variants enables the consolidation of the features of product variants in a domain model [12] and the detection of variability and/or other constraints among product features [24].

The remainder of paper is organized as follows. Section 2 discusses the related work. Section 3 details our approach. Section 4 presents the empirical evaluation of the proposed approach. Section 5 places this work in the bigger context of extractive approach to SPL. Finally, we conclude and summarize possible future research directions.

2 Related work

Goal model [10] is often used in requirement engineering to capture functional (hard) and non-functional (soft) requirements and their dependencies. Hassine et al. [18] applies slicing and dependency analysis to Use Case Map [6] to identify the impact of requirement changes on the system. Zowghi et al. [40] presents a logical framework for modeling and reasoning about the consistency and completeness of the requirements. Lormans [25] developed a methodology to monitor the evolution of requirements and reconstruct requirement traceability.

Feature model [19] is commonly used to represent common and variant requirements in SPL. Thüm et al. [34] utilize SAT solver to classify the evolution of feature model based on how the configurability of the model has changed. Dhungna et al. [12] proposed a model-driven approach to product line evolution. Their approach supports merging of model fragments into a complete variability model as well as the consistency checking and co-evolution of models and architecture.

The Product Feature Model (PFM) in this work is similar to the hard-goal model, as a PFM captures the requirements of a particular product variant. However, our research goal is to support reengineering product variants into SPL. At requirement level, we would like to derive a domain feature model [20], representing configurable requirements in a SPL. In this sense, we refer to our input model as Product Feature Model.

Existing work on requirement and domain engineering has been focused on the modeling, management of, and reasoning about requirements and their evolution. In this work, we present a model-differencing based method to detect and analyze feature changes at requirement level in a family of product variants. This work enables the variability analysis and consolidation of product variants in the task of extractive reengineering into SPL.

Researchers have also investigated the comparison and merging of other models. Xing and Stroulia developed *UMLDiff* [36] for comparing UML class models for supporting evolutionary development of object-oriented software design. Godfrey and Zou [17] use origin analysis to detect the merging and splitting of source-code entities at the file-structure level. Demeyer et al. [11] propose a set of heuristics for detecting evolutions from refactorings by applying lightweight, object-oriented metrics to successive versions of a software system.

Nejati et al. [29] present an approach to matching and merging state chart specifications. Treude et al [35] use a high-dimensional search tree to efficiently compare models that can be represented as direct, typed graphs. GenericDiff used in this work is a general framework for comparing various types of models. In addition to product feature model, we have also applied

GenericDiff to compare the implementation (e.g., Program Dependence Graph) of product variants [38].

Bruntink et al. [5] use clone detection to identify crosscutting concerns, which often implement distinct features. Feature location methods support the recovery of features from product implementations using information retrieval techniques [2][31] or scenario-based dynamic analysis [13][21]. These approaches can be exploited to acquire product feature models from the implementations of product variants.

Alves et al. [1] defines a catalog of feature model refactorings that can be enacted in the extractive reengineering into SPL. They assume that it is known where to apply these refactorings in a software product family. The output of this work can be exploited to identify the opportunities in a family of product variants where such refactorings are applicable.

3 The approach

In this section, we first define the Product Feature Model (PFM) (Section 3.1). We then summarize a catalog of features changes (Section 3.2). We discuss how these changes affect the PFMs lexically and structurally, since we are interested in detecting these changes from their effects on the PFMs. Next, we present *GenericDiff* framework and describe how we configure *GenericDiff* to compare PFMs (Section 3.3). Finally, we analyze the differencing report by *GenericDiff* to infer the changes of product features as product variant evolves (Section 3.4).

3.1 The meta-model of product feature model

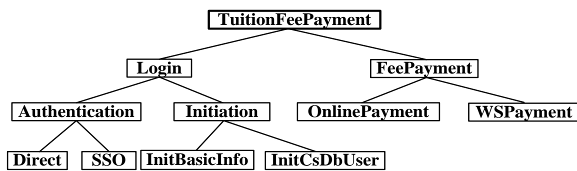

Figure 2. A Partial PFM of WFMS[Shandong]

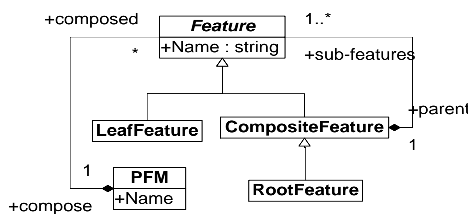

Figure 3. The meta-model of PFM

Figure 2 shows a partial PFM of the product variant WFMS[Shandong] in the WFMS product family [39]. The rectangle nodes denote features. This PFM has a root feature *TuitionFeePayment* that refers to the whole system. The root feature is decomposed into two composite features, *Login* and *FeePayment*. The *Login* is decomposed into two composite features, *Authentication* and *Initiation*, which are further decomposed into leaf features that represent different authentication modes and initiation operations, respectively. The *FeePayment* is decomposed into two leaf features that represent two different payment methods, *OnlinePayment* and *WSPayment.* Such PFMs are the inputs to our approach.

Figure 3 presents the meta-model of product feature model. The meta-model defines rules that must be followed to build correct PFMs. A PFM forms a hierarchy of product features. Each *Feature* in a PFM must be uniquely identifiable by its *name* property. Note that the feature name can be any free-form text that describes the feature. A PFM must have a *RootFeature*, which is a special *CompositeFeature* that represents the corresponding product variant. A *Feature* can be decomposed into sub-features. A feature that has no sub-features is a *LeafFeature*; otherwise it is a *CompositeFeature*. The root feature has no *parent* feature, while a non-root feature must have a *parent* feature, i.e. belongs to a composite feature.

3.2 A catalog of feature changes

We propose a catalog of feature changes that can evolve a PFM. We define four types of atomic changes, namely *rename feature*, *add leaf feature*, *remove leaf feature*, and *move feature*. Furthermore, we define four types of composite changes that can be composed of a sequence of atomic changes: *add feature subtree*, *remove feature subtree*, *split feature*, and *merge feature*. Note that this catalog of feature changes is sufficient to describe the evolution of PFMs. However, it is possible to define other types of feature changes. Our approach can be easily extended to handle the new types of feature changes.

RenameFeature. A consistent naming scheme improves product maintainability, especially when feature names allude to the functions of the product. A feature may be renamed to reflect the underlying implementation changes, the adoption of different technologies, or the changes of application context. Renaming a feature *f* changes the *name* property of the feature *f* in PFM.

AddFeature. A product can be extended with new features. A new leaf feature *nlf* can be added to an existing feature *f*. This creates a new *parent-subfeature* relation from *f* to *nlf*. If *f* is a leaf feature, it becomes a composite feature after the addition. Adding a composite feature and its descendants, i.e., *AddFeatureSubtree* can be achieved by traversing the subtree and adding leaf features in preorder.

RemoveFeature. A product variant may not have some features. An existing leaf feature can be removed

from a PFM. This removes the *parent-subfeature* relation between the removed leaf feature and its *parent* composite feature. A composite feature may become a leaf feature after the removal. Removing a composite feature and its descendants, i.e., *RemovingFeatureSubtree* can be achieved by traversing the subtree and removing leaf features in postorder.

MoveFeature. The feature hierarchy may be reorganized. Moving a feature f (leaf or composite) from a source composite feature *sf* to a target feature *tf* changes the *parent-subfeature* relation between *sf* (*tf*) and *f*. But moving does not affect the *parent-subfeature* relations between f and its sub-features. Moving a feature can be achieved by removing the feature from a PFM and then adding it somewhere else in the PFM. However, we consider *move* as an atomic change, since it better conveys the intention of the change (i.e., the reorganization of feature hierarchy) than the separate addition and removal.

SplitFeature. A feature f can be split into two or more sibling features (including *f*). If f is a composite feature, some of its sub-features will be distributed (i.e., moved) to its new sibling features. Splitting feature can be achieved by first adding new sibling features as leaf features and then moving some of the sub-features of f to the relevant new sibling features.

MergeFeature. Two or more sibling features (including *f*) can be merged into a single feature *f*, as in opposite to splitting a feature. The sub-features of the merged sibling features will become the sub-features of this single feature after the merging. Merging feature can be achieved by firstly moving all the sub-features of *f*'s sibling features to f and then removing these sibling features.

3.3 The differencing of product feature models

Given the PFMs $\{PFM_1..PFM_N\}$ of N product variants, we apply *GenericDiff* to compute pair-wisely the differences between the product feature models PFM_i and PFM_j ($i \neq j; 1 \leq i,j \leq N$) of two product variants. In this section, we give an overview of *GenericDiff* framework. We explain how we configure *GenericDiff* to compare PFMs. The interested reader is referred to [38] for further information about *GenericDiff* framework.

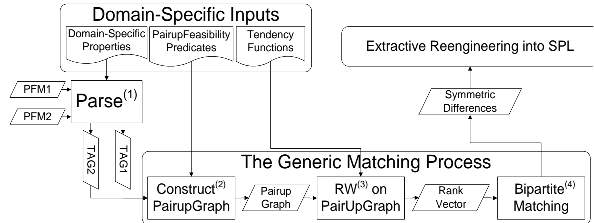

Figure 4. The architecture of *GenericDiff*

Figure 4 shows the architecture of *GenericDiff*. *GenericDiff* takes as input two models to be compared (in this work, i.e., Product Feature Models) and the specifications of domain-specific properties, pairup feasibility predicates, and random walk tendency functions (three concepts to be made clear below) for the comparison of input models. It casts the problem of model comparison as a problem of recognizing the *Maximum Common Subgraph* (*MCS*) of the two Typed Attributed Graphs (TAGs).

Given two PFMs, PFM_1 and PFM_2, *GenericDiff* parses(1) the input models into typed attributed graphs, TAG_1 and TAG_2, according to the meta-model of PFMs (See Section 3.1). The TAG of a PFM forms a *containment tree*, such as the PFM shown in Figure 2. The TAG of a PFM consists of three types of graph nodes, corresponding to *RootFeature*, *LeafFeature* and *CompositeFeature* respectively. Graph edges represent *parent-subfeature* relations between features.

A property of model elements and relations declares a characteristic of their instances. Given a meta-model, one needs to select a set of *domain-specific properties* for each element and relation type that characterize its instances. During the parsing process, *GenericDiff* collects data from the selected properties of each model element and relation and represents them in a characteristic composite vector attribute associated with the corresponding graph node and edge. This composite vector attribute is a compact representation of the properties of model elements and relations for efficient graph indexing and matching.

A feature in PFM has three properties, namely, *name*, a *parent* feature, and a (possibly empty) set of *sub-features*. In the *containment tree* representation of PFM, we define, for a feature node *f*, a composite vector of two atomic vectors. One atomic vector represents the set of words in the *name* property of *f*. We choose the Jaccard coefficient, an efficient and commonly used metric to measure the similarity between two sets of words S_1 and S_2, i.e., $S_1 \cap S_2 / S_1 \cup S_2$. The other atomic vector is a numeric vector that stores the number of the *sub-features* of *f*. Given two such numeric vectors, $[v]$ and $[v']$, we choose Manhattan (Taxicab) distance, i.e., $|v - v'|$, to measure their similarity. The edges of the containment tree of PFM have no characteristic vectors, since they simply represent the *parent-subfeature* relations between features.

Given two TAGs, corresponding to the two compared PFMs, *GenericDiff* constructs(2) a PairupGraph, i.e., a product of the two compared model graphs. The PairupGraph encodes the graph structure of two compared models. A node (edge) of PairupGraph represents a pair of nodes (edges) of two compared model graphs. The construction of PairupGraph is

guided by a set of user-specified *pairup feasibility predicates* to prune the search space according to domain-specific knowledge. For the comparison of PFMs, we simply define the type compatibility of graph nodes, i.e., [*Feature*, *Feature*]. Note that we define the type compatibility in terms of the super-type *Feature*. Therefore, the mappings between graph nodes of different subtypes, such as *LeafFeature* and *CompositeFeature*, are allowed.

The initial distance value of a pair of graph nodes (edges) is calculated as the Euclidean length of the normalized distance vector of the two graph nodes (edges). *GenericDiff* performs a random walk(3) on the PairupGraph, which is an iterative process that propagates the distance values from node pair to node pair based on graph structure. A random walk on a graph can be described by a probabilistic model [26] that is defined by a set of *random walk tendency functions*. For the comparison of PFMs, we use the default random walk settings provided by *GenericDiff* framework that define the random walk tendency functions as linear functions of the distance values of the relevant node and edge pairs.

The random walk on the PairupGraph outputs a rank vector of graph node pairs, each of which is assigned a numerical correspondence measure, i.e., the measure of the quality of the match it represents. *GenericDiff* constructs a bipartite graph from this rank vector of node pairs and selects an optimal matching(4) using Gale-Shapley algorithm [16]. Finally, given a pair of matched graph nodes, *GenericDiff* builds a bipartite graph of their edges and uses Gale-Shapley algorithm again to map their edges.

3.4 Inferring changes to product features

GenericDiff reports a symmetric difference between two compared models. For the comparison of two PFMs, PFM_1 and PFM_2, *GenericDiff* outputs a set *M* of corresponding (i.e., matched) features that exist in both PFMs, a set UM_1 of features that are unique in PFM_1, and a set UM_2 of features that are unique in PFM_2. Based on the differencing report by *GenericDiff*, we developed a tool for automatically inferring feature changes (as defined in Section 3.1) that can evolve PFM_1 into PFM_2 based on the effects of feature changes on the PFMs.

Let *f* be a feature, we define *parent(f)* returns the *parent* feature of *f* and *name(f)* returns the *name* property of *f*. Let $f_1 \in PFM_1$ and $f_2 \in PFM_2$ be a pair of matched features reported by *GenericDiff*, i.e., $(f_1, f_2) \in M$, our tool reports an instance of a particular type of feature change (by tagging (f_1, f_2) with the corresponding change type) as follows:

- if $name(f_1) \neq name(f_2) \Rightarrow < f_1, f_2, Rename >$
- Let $pf_1 = parent(f_1)$ and $pf_2 = parent(f_2)$, if $(pf_1, pf_2) \notin M \Rightarrow < f_1, f_2, Move >$
- Let $sf \in UM_2$, if $parent(f_2) = parent(sf)$ and $\exists (cf_1, cf_2) \in M, f_1 = parent(cf_1)$ and $sf = parent(cf_2)$ and $< cf_1, cf_2, Move > \Rightarrow$ $< f_1, f_2, Split >, < f_1, sf, Split >$
- Let $mf \in UM_1$, if $parent(f_1) = parent(mf)$ and $\exists (cf_1, cf_2) \in M, mf = parent(cf_1)$ and $f_2 = parent(cf_2)$ and $< cf_1, cf_2, Move > \Rightarrow$ $< f_1, f_2, Merge >, < mf, f_2, Merge >$

To detect feature renaming, our tool simply examines the *name* properties of a pair of matched features. To detect feature move, our tool examines whether the parent features of a pair of matched features are matched. To detect feature splitting and merging, our tool essentially examines if some sub-features of a pair of matched features are moved to some unmatched sibling features of this pair of matched features. Note that our tool does not report the splitting/merging of leaf features, since there is no distinction between the effect of splitting a leaf feature and that of adding some new leaf features.

All the pairs of matched features that have not been tagged with the above four types of changes will be tagged with *unchanged*. Finally, all the unmatched features in UM_1 and UM_2 (excluding those tagged with *Split* and *Merge*) are reported as features to be *Removed* and *Added* respectively. If a composite feature and all its descendants are tagged with *Add* (*Remove*), our tool reports an *AddFeatureSubtree* (*RemoveFeatureSubtree*).

4 Evaluation

In this section, we present the empirical evaluation of the proposed approach. More specifically, we investigate two research questions: 1) How accurate is the proposed approach in detecting changes to product features during evolution? 2) How robust is the proposed approach when the PFMs undergo substantial amount of changes during the evolution process?

4.1 WFMS case study

We have applied our approach to analyzing the feature evolution in the product family of WingSoft Financial Management Systems (WFMS). The first product of WFMS was developed in 2003 for Fudan University and it has evolved into a product family with more than 100 customers today. This product family includes 26 product variants. All the product variants share 13 common features, such as *Settlement*, *FileLog*, but also differ in other features specific to a given customer, such as *InitCsDbUser* in $\text{WFMS}^{\text{Shandong}}$ and *SelectByYear* in $\text{WFMS}^{\text{Shanghai}}$.

Among all the WFMS product variants, WFMS$^{\text{Chongqing}}$, WFMS$^{\text{Shandong}}$, WFMS$^{\text{Shanghai}}$ and WFMS$^{\text{Zhejiang}}$ are four major variants. The other product variants have been derived from them with minor changes. Figure 1 shows the evolutionary dependencies among the first product WFMS$^{\text{Fudan}}$ and these four major variants. Each of the four major product variants has on average 30 features and 50KLOC of Java code. The system expert of WFMS product family provided us the PFMs for the four major product variants, which were then pair-wisely compared and analyzed using the proposed model-differencing based approach.

Overall, our approach reported pair-wisely seven or eight feature changes between the four major variants in the WFMS product family. We presented some examples of these feature changes in the introduction section. Our approach reported one wrong feature matching (*SelectByYearOrder*, *SelectByYear*) between WFMS$^{\text{Chongqing}}$ and WFMS$^{\text{Shanghai}}$. These two features are unique features in WFMS$^{\text{Chongqing}}$ and WFMS$^{\text{Shanghai}}$ respectively. However, due to their description and structural similarities, our approach reported them as the "same" feature being renamed. Furthermore, our approach missed one feature mapping (*WSPayment*, *WebServicePayment*) between WFMS$^{\text{Shanghai}}$ and WFMS$^{\text{Shandong}}$. Although the two features represent the similar functionality in two product variants, neither their descriptions nor their hierarchical dependencies with other features are similar enough for them to be recognized as a pair of corresponding features.

The system expert of WFMS found the proposed approach useful in three ways. First, although it is possible for him to manually identify the changes to product features in this small-scale product family, the manual analysis would be ad-hoc and require a high familiarity with the subject product family and its evolution history. In contrast, our approach provides a systematic way to assist him in the analysis of feature evolution in a software product family. Second, our approach recovers the traceability of product features across product variants. This helps to understand variants of existing features and adapt "right" features for reuse in new products. Third, the reported feature changes reveal the inconsistencies of feature descriptions and dependencies in product variants. Understanding and reconciling these inconsistencies is the prerequisite to extractive reengineering for a domain model, representing the common and variant features of a software product family.

4.2 An empirical study with synthesized PFMs

The WFMS case study demonstrates qualitatively the effectiveness of our approach in understanding feature evolution in a family of product variants. However, we would like to further investigate quantitatively how our method scales up to many product variants characterized by many features that have changed over time. Inspired by Thüm et al's recent work [34], in which the synthesized feature models were utilized to evaluate the classification of feature model evolution, we have also designed a controlled experiment to evaluate the performance of our approach with a large volume of synthesized PFMs. This experiment allows us to better understand the strength and weakness of our approach.

4.2.1 The generation of synthesized PFMs

We based our experimentation on the combined feature model of *eShop* [28] and *Home Integration System* (*HIS*) [20] from the feature model repository of S.P.L.O.T [27]. Given this feature model, we developed a tool for generating PFMs in two phases. First, the tool instantiates the feature model to obtain an initial family of PFMs. Next, it iteratively selects a PFM from this family and evolves the PFM by applying the six types of feature changes (as defined in Section 3.2). The evolution is performed according to the user-defined intensity (the number of types of changes being applied) and scope (the percentage of features being changed).

4.2.1.1 The selection of feature models

S.P.LO.T [27] is a benchmark for the research on Software Product Line. It documents a repository of feature models. Feature model [19] is a hierarchy of product features, similar to product feature model. But a PFM represents a concrete product, while a feature model represents all the products in a SPL. Feature model captures the variability among these products in terms of mandatory and optional features, which define the features that must or can be selectively included in a concrete product. It also allows the definition of AND, OR or XOR constraints among the sub-features of a composite feature. AND indicates that all the sub-features of a composite feature must be included in a product as a whole, while OR or XOR indicates that a subset of all the sub-features or only one sub-feature can be included in a product.

We selected the two largest feature models, namely *eShop* and *HIS*. *eShop* has 286 features in total, among which there are 74 mandatory features, 81 optional features and 131 OR-subfeatures under 39 different parent features. There are no XOR-features in *eShop*. *HIS* has 66 features in total, among which there are 44 mandatory features, 10 optional features and 12 XOR-subfeatures under 6 different parent features. We combined the two feature models, *eShop* and *HIS* into one. One may consider that the resulting feature model represents an artificial system that has *eShop* and *HIS* as its two subsystems.

4.2.1.2 The randomized instantiation of PFMs

Given the combined feature model of *eShop* and *HIS*, our date-generation tool first applies a randomized instantiation strategy to generate a family of PFMs from this feature model by randomly selecting features from the root feature down. The instantiation process takes as input three parameters, the size of the initial product family *n* (*n*=10 in this experiment), the probability (α_{OR}) of an optional or an OR-feature to be included in a PFM, and the probability (α_{XOR}) of an alternative feature to be included. In this experiment, we configure α_{OR} and α_{XOR} so that at least 70% of generated PFMs include at least 50% of all the features in the combined feature model of *eShop* and *HIS* [34].

4.2.1.3 The randomized evolution of PFMs

Given an initial family of randomly instantiated PFMs, our data-generation tool then iteratively selects a PFM from this family, evolves it by applying up to six types of feature changes (as defined Section 3.2) according to the user-defined evolution strategies, and adds the evolved PFM back to the PFM family. This process continues until the size of the PFM family reaches the user-defined threshold *K* (*K*=20 in our experiment). This randomized evolution process makes the following assumptions:

- We have studied the feature models listed in S.P.L.O.T repository and found that most feature models define features using phrases, such as "Tuition Fee Payment" in the example given in Figure 2. Thus, we utilize WordNet [14] to mimic the renaming of features. The data-generation tool alters the name of a feature by adding word, removing word, replacing word with synonyms, and reshuffling the order of words.
- The data-generation tool applies only the removal of a leaf feature at a time. The reason is that removing an entire feature subtree usually results in a PFM that has much fewer features than others in the PFM family. This often renders it meaningless to use our approach, since such product variants can be considered as completely different products, while the goal of our work is to detect feature changes in a family of similar product variants.
- The data-generation tool only applies the feature splitting and merging to composite features. Technically, if a leaf feature is split into two leaf features, it is considered as a new leaf feature being added. Similarly, merging two leaf features is considered as removing one of the leaf features.

The evolution of a PFM is performed according to the user-defined intensity (the number of types of changes being applied) and scope (the percentage of features being changed). More specifically, we have designed two evolution strategies: feature-centric and change-type-centric. With the feature-centric strategy, one can specify the percentage of features that will be changed during the evolution process, but the types of changes being applied to each changed feature are freely chosen. In contrast, with the change-type-centric strategy, one can specify several types of changes that will be applied to the features of a PFM, but each type of changes can be applied to different sets of randomly chosen features. In the next section, we discuss how we apply these two evolution strategies in our experiment.

4.2.2 The performance of our technique

The data-generation tool generates a family of PFMs, which are similar but also different from each other. We randomly select *S* (*S*=10 in our experiment) pairs of PFMs from this PFM family and apply the approach presented in Section 3 to detect the feature changes between them. The data-generation tool records the change history that the PFMs have undergone during the randomized evolution process. This change history serves as an oracle to evaluate the accuracy and robustness of our approach.

Given two PFMs, PFM_1 and PFM_2, we denote the recorded change history (i.e., expected changes) and the detected feature changes by our approach (i.e., reported changes) as M_E={<f_1, f_2, *changetype*>} and M_R={<f_1, f_2, *changetype*>} respectively, where f_1 refers to a feature in PFM_1, f_2 is the corresponding feature of f_1 in PFM_2, and *changetype* is the type of changes between the two features. The *changetype* can be *unchanged*, *rename*, *move*, *split*, or *merge* (See Section 3.2 and Section 3.4). Since our objective in this experiment is to evaluate the accuracy of our approach in identifying corresponding features in two PFMs, we omit the addition and removal of features. We evaluate the accuracy of our approach in terms of the precision and recall: precision *P* is the percentage of correctly reported changes, i.e., $M_R \cap M_E / M_R$ and recall *R* is the percentage of changes reported, i.e., $M_R \cap M_E / M_E$.

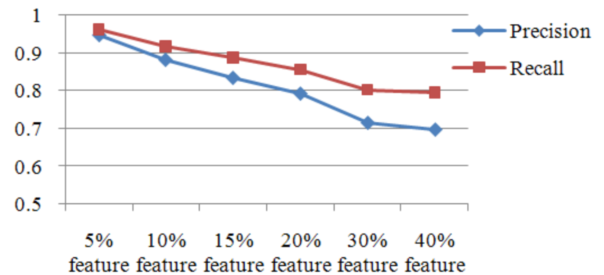

Figure 5. The precision and recall for change-type-centric strategy

Figure 5 summarizes the precision and recall of our approach in an experiment in which a family of PFMs has been evolved according to change-type-centric strategy. In this experiment, all six types of changes

(see Section 3.2) have been applied during the evolution process. We incrementally increased the scope, i.e., the percentage of features that will be changed by each type of changes, from 5% to 40%. We compute the precision and recall of our approach in *S* (*S*=10 in this experiment) comparisons and then take the arithmetic average value as reported in Figure 5.

The accuracy of our approach degrades as the scope of changes increase. It seems that it hits the bottom at the scope of 30%, which is really a very intensive evolution. Since each of six types of changes have been applied to 30% of randomly chosen features, each feature of this PFM statistically undergoes about 1.8 times of different types of changes on average. If such intensive evolution happens to a real-world system, the original product and the resulting product would be deemed as two completely different products.

We have conducted another experiment in which a family of PFMs has been evolved according to feature-centric strategy. Figure 6 and Figure 7 summarizes the results. In this experiment, we first decide the scope of features to be changed and then increase the intensity of changes from one type of change to six types of changes. We run this experiment at four increasing scopes, i.e., 10%, 20%, 30% and 40%. In general, the precision and recall of our approach drop as the intensity and scope of changes increase. But it still recovers 86% of all the corresponding features at the precision of 81% in the worst scenario, in which 40% of features have been changed, each of which suffers three randomly chosen types of changes.

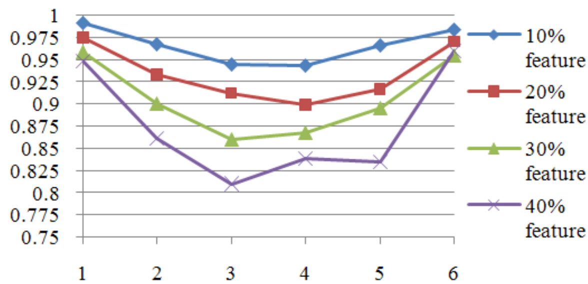

Figure 6. The precision for feature-centric strategy

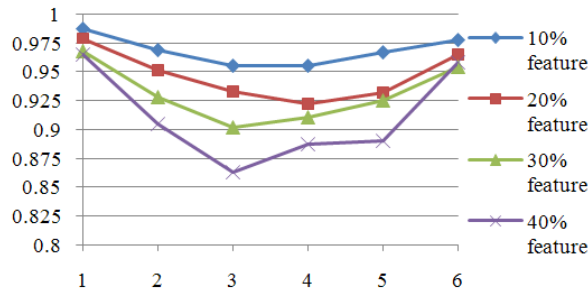

Figure 7. The recall for feature-centric strategy

One interesting observation in Figure 6 and Figure 7 is that when the scope (i.e., the percentage of feature to be changed) is fixed, the precision and recall of our approach by randomly applying five or six types of changes are actually better than that of applying three or four types of changes. We manually inspected the experimental data and found out that this is resulted from the removal of features. At the intensity of five or six types of changes, it is highly likely that the type of feature removal will be applied during the evolution process. When a feature is removed, all the changes that have already been made to it will be lost. Thus, feature removals actually simplify the comparison.

With feature-centric strategy, the scope of changes is fixed, which means that the rest of the features remain unchanged. In this case, even a simple name-based matching approach can recover at least *1-p%* (let the scope of changes be *p%*) of all the corresponding features. We comparatively study our approach against the simple name-based matching approach. Overall, our approach can recover more than 60% of the corresponding features that are missed by simple name-based matching. In a case in which 80% of features have been moved and/or renamed, the name-based matching only reports 27% of all the corresponding features, while the recall of our approach is 67%. This is because our approach identifies corresponding features based on not only their names but also their structural context.

Overall, our approach is able to produce an accurate change reports between the PFMs of product variants, even the PFMs have undergone intensive evolution. We manually inspected our experimental data and identified two main causes of false positive (i.e., erroneously reported) changes and false negative (i.e., missed) changes using our approach.

First, it is difficult for our approach to determine the correspondences between features with little or very similar structural context (i.e., dependencies and relationships with other features). For example, the leaf features become an issue, since they have no structural information other than their name property. Consider an example from *eShop* system. One product variant has a leaf feature *Internal Tracking*, while the other has a leaf feature *Partner Tracking*. When comparing the PFMs of these two products, our approach reports that *Internal Tracking* and *Partner Tracking* are the "same" feature being renamed in different product variants. However, *Internal Tracking* and *Partner Tracking* are actually two alternative ways of shipment tracking, which should not be considered as the "same" feature.

Second, if product features suffer various types of changes at the same time, for example, a feature *f* is moved to another composite feature, and then split into

several features, finally it is renamed, and our approach may not recognize the correspondences between the origin feature *f* and the resulting feature, since their name properties and structure changed dramatically. Since we keep track of the change history of PFMs in the data-generation process, we consider such cases as missed changes. However, a human expert may deem such two features as two different features since they have been changed dramatically, even though one feature is the origin of the other.

5 Applications

Having evaluated the quality and robustness of our approach, the next question we would like to address is "what is this good for?" In this section, we place the work presented in this paper in the overall context of our research on extractive reengineering into SPL. We will briefly discuss three applications based on the results of this work. These applications are currently under development – to a different degree of maturity.

The long-term objective of our research is to support reengineering a family of similar product variants into a SPL for systematic reuse. Figure 8 depicts the overall methodology we have adopted for our work. The input to our methodology is the software artifacts of product variants at different levels of abstraction, for example, product feature model at requirement level, UML class model at architecture and design level, and Program Dependence Graph (PDG) at implementation level. The output of our reengineering methodology is a collection of core assets of a SPL, which may include domain feature model (FM) [19], Product Line Architecture (PLA) [9] and generic components (GenericComp) [32].

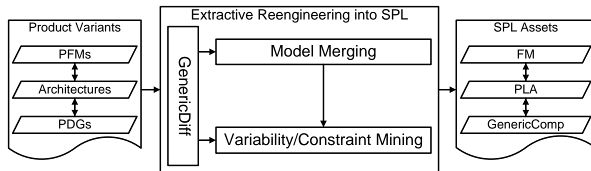

Figure 8. Reengineering Product Variants into SPL

The successful recovery and understanding of commonalities and differences among the artifacts of product variants strides the first step towards our research objective. In this paper, we presented our application of *GenericDiff* to detect changes to product features at requirement level. In our onging work, we have also applied *GenericDiff* to compare software artifacts of product variants at design and implementation level. For example, in [37], we combined clone detection technique and *GenericDiff* for the similarity analysis of software system. The implementation of software product is represented as Program Dependence Graph [15]. Our approach is able to efficiently and accurately identify similar reusable code fragments and highlight their semantic differences.

Based on the differences *GenericDiff* reports, we are currently investigating the use of model merging and data-mining techniques to reverse-engineer the configurable domain feature model, product line architecture and generic components. For example, we are using description logic [3] to model and reason about the conflicts and inconsistencies among product feature models and investigating the merging of PFMs of product variants using graph transformation tools, such as [33]. In addition to model merging, we are also investigating the recovery of the variability and other general constraints between product features by mining association rules [7] or subtree patterns [8] from the differences among the PFMs of product variants.

Last but not least, recovering traceability in software artifacts of product variants across requirement, design and implementation can provide important insights into the development and maintenance of a SPL. Such traceability is also essential in the derivation of concrete products from a SPL. Researchers have investigated the use of information retrieval [2], scenario-based dynamic analysis [13], or the combination of both [31] for the recovery of traceability. However, existing work on traceability recovery analyzes only artifacts of a single software system. In our problem setting, we have a family of similar product variants, which should be exploited.

The ability to compare and identify the differences in a family of product variants at different levels of abstraction can assist the task of traceability recovery. The underlying intuition is that the presence or absence of a feature in a product variant should be reflected in the presence or absence of certain design elements and code fragments. In our ongoing work, we are combining the information retrieval techniques with software differencing results to recovery feature–design element–code fragment traceability in a product variants family.

6 Conclusion and future work

In this paper, we presented our approach to understand feature evolution in a family of software product variants. We entail that features and their dependencies for each product variant are documented as product feature model. The innovation of our approach is to exploit model differencing technique (*GenericDiff*) to detect evolutionary changes to product features at requirement level. Based on the differences between product feature models as reported by *GenericDiff*, our approach automatically infers evolutionary changes that occurred to product features of different product variants.

We evaluated the effectiveness and scalability of our method using a real-world product family of financial systems as well as a large volume of systematically synthesized data. We showed that our method yields good results and scales to large systems.

In the future, based on the results of this work, we plan to investigate merging techniques to support the (semi-)automatic reconciliation of inconsistent product feature models. Furthermore, we also plan to exploit data mining techniques to discover the variability and other general constraints among product features. These techniques will lead to further automation of extractive reengineering of a family of similar product variants into a SPL.

Acknowledgements

This work was supported by the Ministry of Education of Singapore (AcRF Tier 1 R-252-000-399-112 and R-252-000-422-112).

References

[1] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P. and Lucena, C.J.P.: Refactoring product lines. GPCE 2006: 201-210

[2] Antoniol, G. and Guéhéneuc, Y.G.: Feature Identification: An Epidemiological Metaphor. IEEE Trans. Software Eng. 32(9): 627-641 (2006)

[3] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D. and Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, Applications. Cambridge University Press, Cambridge, UK, 2003

[4] Benavides, D., Martin-Arroyo, P.T. and Cortes, A.R: Automated Reasoning on Feature Models. CAiSE 2005: 491-503

[5] Bruntink, M., Deursen, A., Engelen, R. and Tourwé, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE Trans. Software Eng. 31(10): 804-818 (2005)

[6] Buhr, R. J. A. and Casselman, R. S.:Use case maps for object-oriented systems. Prentice Hall, 1996.

[7] Buneman, P. and Jajodia S.: Mining Association Rules between Sets of Items in Large Databases, Washington, D.C., 1993.

[8] Chi, Y., Muntz, R., Nijssen, S. and Kok, J.N.: Frequent Subtree Mining - An Overview, Fundamenta Informaticae, v.66 n.1-2, p.161-198, January 2005

[9] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

[10] Dardenne, A., Lamsweerde, A. and Fickas, S.: Goal-Directed Requirements Acquisition. Science of Computer Programming, 20(1-2), April 1993

[11] Demeyer, S., Ducasse, S. and Nierstrasz, O.: Finding refactorings via change metrics. OOPSLA 2000: 166-177

[12] Dhungana, D., Neumayer, T., Grünbacher, P. and Rabiser, R.: Supporting Evolution in Model-Based Product Line Engineering. SPLC.319-328, 2008

[13] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code. IEEE Transactions on Software Engineering, vol. 29, no. 3, pp. 210 - 224, March 2003.

[14] Fellbaum, C.: WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press, 1998.

[15] Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3): 319-349, 1987.

[16] Gale,D. and Shapley, L.S.: College admissions and the stability of marriage. American Mathematical, 69:9-14, 1962.

[17] Godfrey, M. and Zou, L.: Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. IEEE Trans. Software Eng. 31(2): 166-181 (2005)

[18] Hassine, J., Rilling, J., Hewitt, J. and Dssouli, R.: Change Impact Analysis for Requirement Evolution using Use Case Maps, Proc. of the 8th Int. Workshop on Principles of Software Evolution, pp. 81-90, 2005.

[19] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S., Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, November 1990

[20] Kang, K.C., Lee, J. and Donohoe, P.: Feature-oriented project line engineering. IEEE Software., 19(4):58-65, July 2002.

[21] Koschke, R. and Quante, J.: On dynamic feature location. ASE 2005: 86-95

[22] Krueger, C.W.: Easing the Transition to Software Mass Customization. In: Proc. of the 4th International Workshop on Product Family Engineering, October 2001, pp. 282–293 (2001)

[23] Krueger, C.W.: Practical Strategies and Techniques for Adopting Software Product Lines. ICSR 2002,349-350

[24] Liebig, J., Apel, S., Lengauer, C., Kästner, C. and Schulze, M.: An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. ICSE 2010,pp: 105–114.

[25] Lormans, M.: Monitoring requirements evolution using views. CSMR'07 , 349–352. 2007

[26] Lovasz, L.: Random walks on graphs: A survey. 1993

[27] Mendonca, M., Branco, M. and Cowan, D.: S.P.L.O.T. - Software Product Lines Online Tools. OOPSLA 2009

[28] Mendonca, M., Wasowski, A., Czarnecki, K. and Cowan, D.: Efficient Compilation Techniques for Large Scale Feature Models. In Proc. Int'l Conf. Generative Programming and Component Engineering, pages 13–22. ACM, 2008

[29] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M. and Zave, P.: Matching and Merging of Statecharts Specifications. ICSE'07, 54–64. 2007

[30] Ommering, R.C.: Building product populations with sofware components. ICSE 2002: 255-265

[31] Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G. and Rajlich, V.: Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Trans. Software Eng. 33(6): 420-432 (2007)

[32] Rajapakse, D.C. and Jarzabek, S.: Towards generic representation of web applications: solutions and trade-offs. Softw., Pract. Exper. 39(5): 501-530 (2009)

[33] Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P.: Automated Merging of Feature Models Using Graph Transformations.GTTSE 2007, Braga, Portugal, July 2-7, 2007.

[34] Thüm, T., Batory, D., and Kästner, C.: Reasoning about edits to feature models. ICSE 2009: 254-264

[35] Treude, C., Berlik, S., Wenzel, S. and Kelter, U: Difference computation of large models. ESEC/FSE 2007: 295-304

[36] Xing, Z. and Stroulia, E: Refactoring Detection based on UMLDiff Change-Facts Queries. WCRE 2006: 263-274

[37] Xing, Z., Xue, Y. and Jarzabek, S.: How clones are different semantically? Tchnical Report, School of Computing,NUS,2010.

[38] Xing, Z.: GenericDiff: A General Framework For Model Comparison. Technical Report, School of Computing,NUS,2010

[39] Ye, P., Peng, X., Xue, Y. and Jarzabek, S.: A Case Study of Variation Mechanism in an Industrial Product Line. ICSR 2009: 126-136

[40] Zowghi, D. and Offen, R.: A logical framework for modeling and reasoning about the evolution of requirements. RE97. 247–259. 1997