KIPS Tr. Software and Data Eng.
Vol.6, No.7 pp.337~352 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2017.6.7.337

A Feature-Oriented Method for Extracting a Product Line Asset from a Family of Legacy Applications

Hyesun Lee† · Kang Bok Lee††

ABSTRACT

Clone-and-own reuse is an approach to creating new software variants by copying and modifying existing software products. A family of legacy software products developed by clone-and-own reuse often requires high maintenance cost and tends to be error-prone due to patch-ups without refactoring and structural degradation. To overcome these problems, many organizations that have used clone-and-own reuse now want to migrate their legacy products to software product line (SPL) for more systematic reuse and management of software asset. However, with most of existing methods, variation points are embedded directly into design and code rather than modeled and managed separately; variation points are not created ("engineered") systematically based on a variability model. This approach causes the following problems: it is difficult to understand the relationships between variation points, thus it is hard to maintain such code and the asset tends to become error-prone as it evolves. Also, when SPL evolves, design/code assets tend to be modified directly in an ad-hoc manner rather than engineered systematically with appropriate refactoring. To address these problems, we propose a feature-oriented method for extracting a SPL asset from a family of legacy applications. With the approach, we identify and model variation points and their relationships in a feature model separate from implementation, and then extract and manage a SPL asset from legacy applications based on the feature model. We have applied the method to a family of legacy Notepad++ products and demonstrated the feasibility of the method.

Keywords : Extractive Approach to Software Product Line Engineering, Feature-Orientation, Legacy Applications, Copy-and-Own Reuse

레거시 어플리케이션 제품군으로부터 제품라인 자산을 추출하는 휘처 기반의 방법

이 혜 선† · 이 강 복††

요 약

복제 및 소유(Clone-and-own) 재사용은 기존의 소프트웨어 제품을 복사하고 수정하여 새로운 소프트웨어를 개발하는 방법이다. 복제 및 소유 재사용으로 개발된 레거시 소프트웨어 제품군은 일반적으로 리팩토링 없이 패치 업 되고 구조적으로 저하되기 때문에 높은 유지보수 비용을 필요로 하고 오류가 발생하기 쉬운 경향이 있다. 기존에 복제 및 소유 재사용 방법을 사용했던 많은 회사들이 이러한 문제를 해결하고 소프트웨어 자산을 더 체계적으로 재사용하고 관리하기 위하여 레거시 제품들을 소프트웨어 제품라인으로 전환하려고 하고 있다. 하지만 대부분의 기존 방법들은 가변점(Variation points)을 디자인과 코드로부터 분리해서 모델링하고 관리하지 않고 디자인과 코드에 바로 임베드시킨다. 즉, 가변점이 가변성 모델을 기반으로 체계적으로 생성되고 관리되지 않는다. 이러한 기존 방법들은 다음의 문제를 야기한다. 기존 방법에서는 가변점 간 관계를 이해하기가 어렵기 때문에 가변점이 임베드 된 코드를 유지보수하기가 어렵고 코드가 변경 및 진화될 때 오류가 생기기 쉽다. 또한 소프트웨어 제품라인이 진화할 때 디자인/코드 자산이 적합한 리팩토링을 적용하여 체계적으로 변경되는 것이 아니라, 애드 혹(Ad-hoc) 방식으로 직접적으로 변경되는 경향이 있다. 본 논문에서는 이러한 문제를 해결하기 위하여 레거시 어플리케이션 제품군으로부터 소프트웨어 제품라인 자산을 구축하는 휘처 기반의 방법을 제안한다. 제안하는 방법에서는 가변점과 가변점 간 관계를 식별하고 이들을 구현으로부터 분리하여 휘처 모델로 모델링한다. 그리고 휘처 모델을 기반으로 레거시 어플리케이션으로부터 소프트웨어 제품라인 자산을 추출하고 관리한다. 제안하는 방법을 레거시 Notepad++ 제품군에 적용을 하여 방법의 실행가능성을 검증하였다.

키워드 : 소프트웨어 제품라인 공학에 대한 추출식 접근법, 휘처 기반, 레거시 어플리케이션, 복제 및 소유 재사용

※ This work was supported by ETRI R&D Program grant funded by the Korea Government (MSIP) [17ZH1310, Development of infra-less PDR based connected helmet system for augmented cognition], and supported by the Fire Fighting Safety & 119 Rescue Technology Research and Development Program funded by the Ministry of Public Safety and Security [16AC3200, Development Smart Helnet for Fireman].

† 정 회 원 : 한국전자통신연구원 초연결통신연구소 IoT연구본부
†† 비 회 원 : 한국전자통신연구원 초연결통신연구소 IoT연구본부
Manuscript Received : April 11, 2017
Accepted : April 29, 2017
* Corresponding Author : Hyesun Lee(hyesun.lee@etri.re.kr)

1. Introduction

When software organizations develop new software products similar to the ones that they had developed before, they typically use clone-and-own reuse. Clone-and-own-reuse [1] is an approach to creating new software variants by copying and modifying existing software products. With the approach, each newly created software variant assumes its own maintenance trajectory separated from that of the existing products. Many organizations still use this approach when creating products with new features. (Organizations such as Nokia [2], Danfoss Drives [3], and Hitach [4] have mentioned in publications that their earlier products had been developed in this way.)

The clone-and-own reuse often leads to a number of software variants with duplicated code that need to be maintained separately, thus the maintenance cost increases at an alarming rate [4]. Moreover, software developed using the clone-and-own approach tends to be error-prone because of complex "invisible" dependencies created at the code level [5]. To overcome these difficulties and keep a family of products under control, many organizations that have used clone- and-own reuse now want to migrate their legacy products to software product line (SPL).

Practitioners and researchers [2-15] have published their experiences of extracting a SPL asset[1] from a family of legacy products developed using clone-and-own reuse. However, most of these publications described organization or domain specific experiences rather than introducing a systematic method that can be repeated in different organizational or domain contexts.

Some papers proposed systematic methods for identifying variation points and variants between legacy software variants and reengineering the legacy variants to a reusable asset. For example, Bayer and others ([6-7]) used a product map and a decision table to identify variation points and variants, and then they reengineered legacy design/code and embedded the variation points/variants into an asset using a wrapping scheme ([6]) and a conditional compilation technique. Alves and others ([8-10]) identified variation points in terms of aspects and then used aspect-oriented refactoring to embed the aspects into code. These methods, however, did not model variation points separate from the code. This may cause the following problems: First, relationships between variation points are implicit in the design/code, thus it is difficult to understand the relationships between the variation points. Therefore, it is hard to maintain such code, and the code tends to become error-prone as it evolves. Also, when a SPL evolves, design/code assets are directly changed and it is difficult to analyze and understand "the evolution of variation points/variants"; evolution is "silent." Without understanding the "trends of evolution," asset management tends to be reactive rather than proactive thus losing an opportunity to reflect anticipated changes to the asset while refactoring. We need to create a variability model separate from the SPL asset when we extract an asset from legacy products, analyze the variability model, and then systematically refactor and embed variation points into the asset based on the variability model.

If we have a variability model separate from implementation, however, we need to manage and maintain traceability and consistency between the variability model and the design/code; when the variability model is changed, we should be able to trace to the corresponding variation points and variants embedded in the asset easily. Also, we should be able to analyze the consistency between the variability model and the variation points/variants embedded in the asset code when a new variation point or a variant is added, or existing ones are changed, and vice versa.

To address these issues, we introduce a feature-oriented approach to SPL in this paper. We used legacy products of Notepad++ [16] to illustrate the concept of our method. It is a small example but we believe that it has enough complexity to expose the key aspects of our method and demonstrate the feasibility.

A family of legacy Notepad++ products is briefly introduced in section 2. Then, we introduce the underlying concepts of our method in section 3, followed in section 4-6 by a discussion of our method with illustrative examples using the legacy Notepad++ products. We evaluate our method in section 7. Section 8 includes related works and then we conclude this paper in section 9 with a discussion and future works.

2. Background on NOTEPAD++

Notepad++ [16] is a popular open-source code editor written in C++ language. The primary and basic services of Notepad++ are file management (e.g., file open, save), source code editing (e.g., code copy/paste, indentation), and editing-view control (e.g., zoom in/out editing-view, creating new editing-view). Notepad++ supports editing of programs in various programming languages (e.g., C/C++, Java, XML) and encoding standards (e.g., ANSI, UTF-8, UCS-2).

1) A SPL asset includes requirement specification, design, code, test cases and any other artifact used to develop and maintain SPL

The development history of Notepad++ shows a single evolutionary path. Since Notepad++ was first released in November 2003, new features (such as bookmark, user-defined language support, GUI configuration, etc.) have been added , and about 100 versions of Notepad++ have been released until now.

We identified the following problems in the long evolutionary path of Notepad++: Through the evolution, a number of features have been added, so the current version of Notepad++ has hundreds of features including some features (e.g., Google/Wikipedia search) that are helpful but not necessary for source code editing. Some customers may use all features, but others may want only primary features necessary for source code editing. Also, some features requested by the users with specific needs caused performance degradation, and there have been complaints at the user forum [17]. For example, there was a request for counting exact characters in UTF-8 continuously and this feature was added to version 5.7. However, this feature caused a major performance degradation; The version 5.7 of Notepad++ requires about 2 seconds to complete any action on 30MB file when in UTF-8.

The feature-oriented extractive approach to product line engineering can address these problems; the features that are helpful but not necessary for source code editing can optionally be included in a product according to customer needs.

A set of legacy Notepad++ versions is used in section 5 and 6 to illustrate the method, and the evaluation results will be discussed in section 7. We will introduce underlying concepts of the method in the following section.

3. Underlying Concepts

In this section, we discuss the rationales behind our approach.

Since feature modeling [18] was introduced in 1990, the concept of “feature” has been used in many researches/practices in software product line engineering as the unit of: capability/service that is delivered to customers; parameterization for asset components; development and delivery to customers; product configuration and configuration management; and product management for targeting specific market segments [19]. Therefore, feature model can be used as a central model for representing and managing variability and for extracting and maintaining a SPL asset; that is, variability of a family of legacy products is identified in terms of features, relationships between the features are modeled in a feature model, the legacy products are re-engineered to SPL assets with embedded variation points using optional and alternative features of the feature model, and then the assets are configure and managed systematically based on the feature model.

In this paper, we propose a feature-oriented extractive method that uses a feature model as the central model for variability management. The underlying concepts of the approach are as follows (We assume that each of the legacy products has been developed using the clone-and-own reuse and managed like many single products, and variation points are not embedded in the products.):

(1) Separation of a variability model from its implementation. The problems of existing approaches are due to direct embedment of variation points into design and code without a separate and concurrently managed variability model as discussed in section 1. To overcome these problems, we create a variability model (as a feature model) separate from its implementation.

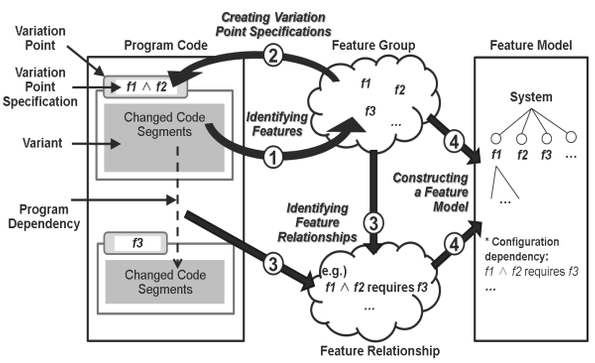


Fig. 1. Separation of a Variability Model from Asset Code

In our approach, we compare legacy products to understand the differences and extract variants and associated variable features (① in Fig. 1). We consider differences as variants, and create a variation point for each variant and assign a logical expression with features as operands; we call this expression a variation point specification (② in Fig. 1). If a feature selection made for a product satisfies the specification of the variation point, the variant associated with it is included in a product.

Then we analyze if there are any program dependencies between variants. For any two variants that have program dependencies, we define features relationships for the features used in the variation point specifications of the related variants (③ in Fig. 1). The feature model (created by ④ in Fig. 1) is used to embed variation points and variants into the asset code (⑤ in Fig. 2) using the variability mechanism (e.g., macro processing) provided by

the programming language; the variation points defined in step 2 will be refined in steps 5 to 7 based on the feature model.

(2) Embedment of variation points and variants in the asset that is consistent with the feature model. We often need to reengineer the legacy code to turn it into reusable asset that is structurally and operationally consistent with and also traceable to and from the feature model (See ⑥ and ⑦ in Fig. 2.). That is, the structural relationships between variants (also between associated variation points) of the asset should be consistent with structural relationships (i.e., composed-of and generalization-specialization) between features of the feature model. Also, call dependencies between variants should be consistent with configuration dependencies (i.e., require and exclude) between features of the feature model. For example, in Fig. 2, a structural relationship between the variation points/variants (mapped to f1 and f11) of the asset code (in the figure on the right) should be consistent with a composed-of relationship between f1 and f11 of the feature model (in the figure on the left).

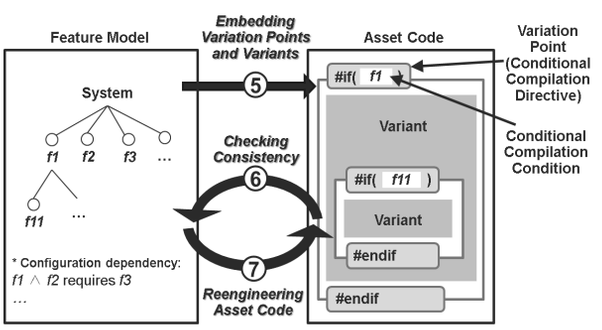


Fig. 2. Embedment of Variation Points/Variants in the Asset that is Consistent with the Feature Model

This feature-oriented extractive approach has the following advantages:

- As relationships between variable features are explicitly identified and modeled, we can easily understand the relationships between the features and associated variation points and variants embedded in the asset, and, therefore, can more systematically develop and maintain the asset than otherwise.
- As the structure of the asset is consistent with that of the feature model, it is easy to trace to features embedded as variation points in the asset. Also, it is easy to predict impacts of feature changes; when a feature changes, we can identify the asset code that may have to be modified together by tracing to all features (and their implementations) that are related with the changed feature.
- The feature model can be used as a product configuration model. Configuration tools can enforce configuration rules following feature dependencies by, for example, pruning the features that can not be selected along with the features that have already been selected because of "exclude" relationships between them.

Based on these ideas, the method processes were defined, which are outlined and explained in the following section.

4. Feature-oriented Method: Strategies of the Method

In this section, we first discuss the challenges addressed in this research, and describe the strategies we adopted to address the challenges. Then we introduce our method with a detailed explanation.

To apply the concepts introduced in section 3, the challenges that needed to addressed are as follows:

- (1) To completely identify variation points and variants of a family of legacy products, we need to compare every combination of the products, but the number of comparisons will increase exponentially as the number of legacy products increases, which makes our method not scalable.
- (2) As features are used to insert variation points in the asset components extracted from a family of legacy programs, we need to determine the right level of granularity of features that correspond well to differences between legacy programs.
- (3) Structural relationships and configuration dependencies between features are hidden in the legacy programs rather than explicitly captured in a document. Also, some features could be omitted or scattered in the implementation, which makes identifying relationships between features difficult.
- (4) Checking consistency between variation points of an asset and the feature model is a time consuming and costly task. (For example, automotive software assets usually have thousands of features and more than 10 million lines of code [20]; a tremendous amount of time and effort are required for consistency checking.)

To address the challenge (1), we have decided to make pair-wise comparison along the clone-and-own path(s)

(Fig. 3) of the family of legacy products. We can identify all additions, deletions, and changes, i.e., feature variations with this approach. The rationale of this approach is as follows: If the program B is directly derived from A, and C is directly derived from B, then differences (i.e., added/deleted code segments) between A and C are a subset of differences between A and B and those of B and C. Also, if the program F and G are directly derived from E, then differences between F and G are a subset of differences between E and F and those between E and G.

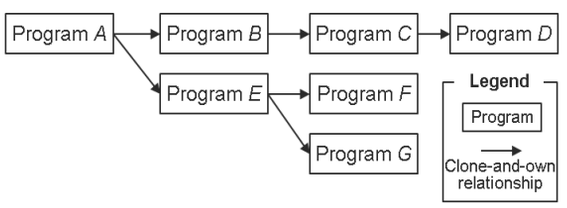


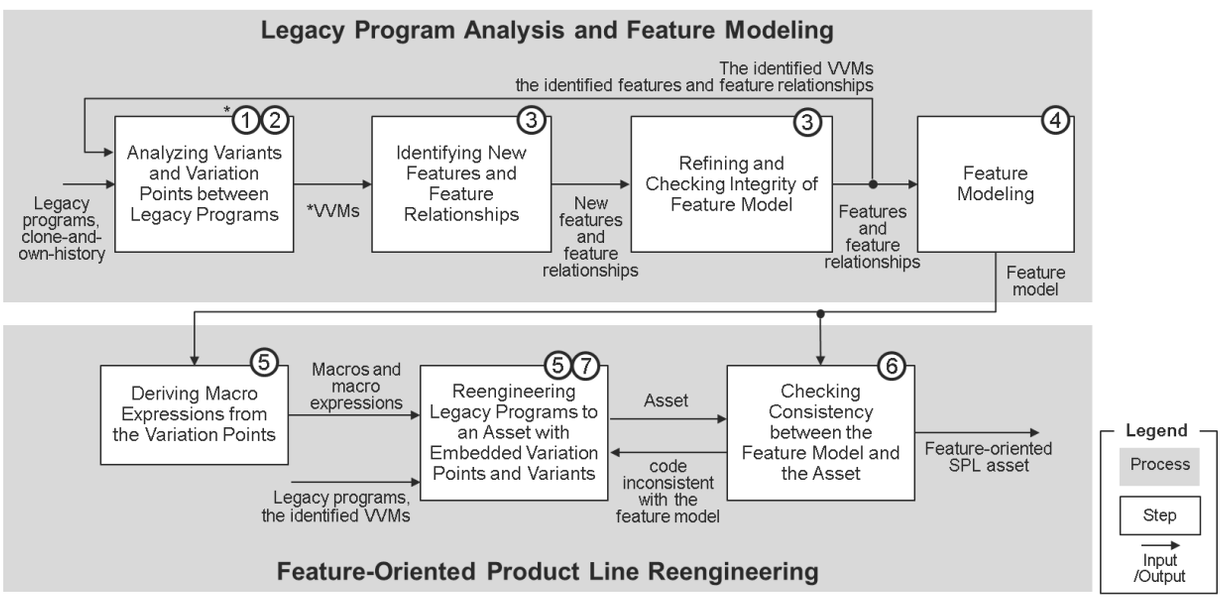
Fig. 3. An Example of Clone-and-Own Paths

Fig. 3 shows an example of clone-and-own path of seven legacy programs, and an arrow between programs shows a clone-and-own relationship (e.g., the arrow from B to C indicates that C is directly derived from B.). Each pair of programs that are adjacent to each other in a clone-and-own path is analyzed in sequence, and we need (n-1) comparisons for the n number of legacy programs for a complete analysis. For the example in Fig. 3, we need to analyze the following six pairs in order: A-B, B-C, C-D, A-E, E-F, and E-G for a complete analysis.

The proposed approach could significantly reduce the number of comparisons from exponential to linear. For the SPL with a large number of legacy programs, a complete analysis may be too costly. Of these pairs, we may choose to select pairs close to the end of each path. Those features that are not present in the recent products may no longer be useful; here, we need domain experts' opinion.

To address the challenges (2) and (3), we propose a systematic method and rules for constructing a feature model bottom-up by comparing the legacy program code. Existing researches in feature-oriented product line engineering typically construct a feature model top-down from requirements. However, features identified by a top-down approach usually capture functionalities/services in abstraction and, when an abstract feature is mapped to code, they tend to cut across many code units (i.e., module, component). With a bottom-up approach, we can identify more concrete low level features (than a top-down approach) each of which corresponds well to a variant. Moreover, with the bottom-up approach, we could identify variable features and feature relationships that are not explicit in documentation.

To address the challenge (4), we defined rules for checking consistency between variation points embedded in the asset and the feature model (The rules will be explained in section 6.3.). These rules can be checked using a CASE tool, reducing effort for the consistency checking substantially.



* The circled numbers correspond to the circled numbers in Figure 1 and Figure 2
* **VVM:** Variation-point-and-variant mapping

Fig. 4. An Overview of the Method

Based on these strategies, we defined a method consisting of two engineering processes that can proceed iteratively and incrementally (as shown in Fig. 4): analyzing the family of legacy programs and constructing a feature model bottom-up; and reengineering the legacy programs and creating an asset that are consistent with the feature model. Note that in this paper we focus on variable (i.e., optional or alternative) features and their relationships rather than mandatory features[2].

Details of the processes and artifacts from each activity of the processes are discussed in section 5 and 6.

5. Feature-oriented Method: Legacy Program Analysis and Feature Modeling

The first process (the upper part of Fig. 4) consists of the steps for creating a feature model from a family of legacy programs as shown in Fig 1 (The circled numbers in Fig. 4 correspond to the circled numbers in Figs. 1 and 2.). These steps are semi-automatic, and the steps 1 through 3 of the process are performed iteratively and incrementally following the clone-and-own path(s) of a family of legacy programs.

Each step of the process is explained in the following subsections.

5.1 Analyzing Variants and Variation Points

To identify variable features and their relationships, we first identify differences between legacy program code and determine which differences are related to which features.

We use an abstract syntax tree (AST) based program comparison [21] to automatically identify code changes between two legacy programs. With the AST-based comparison, we can compare structures of two programs and get more precise results than with a lexical comparison. (A lexical comparison identifies code changes at the lexical level ignoring high-level design changes, so it often provides results that are logically incorrect.). In addition, we can also identify program dependencies from a reversed AST, which are used in the next step.

After identifying code differences, we use domain knowledge to determine which code changes are related to which features (Note that this is not an automatic task). If a code change is related to addition of a new feature(s) or removal of an existing feature(s), we consider the changed code as a variant and name the variable feature(s). A logical expression with these feature(s) as operands can specify a variation point to the variant (See Fig. 1 for the relationship between variation points, variants, and features.). We represent the relationship between the code change (i.e., variant) and the variation point using variation-point-and-variant-mapping. (We call this mapping as "VVM" in short.) We define VVM as follows.

Definition 1. Variant (or Code Variant):

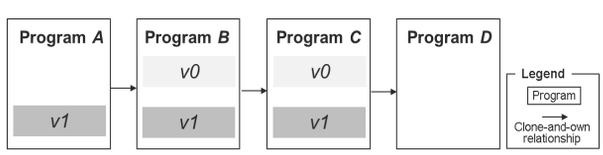
- A variant is a code segment that is different between ASTs of two legacy programs. The unit of variant is one of file, class[3], method, statement, or variable.

Definition 2. Variation Point:

- A variation point is specified by a logical expression with features as operands. Logical value of a feature is true if it is selected and false otherwise.

Definition 3. VVM (Mvp-v):

- Let VP be a set of variation points and V be a set of variants.
- A set of mappings between variation points and variants, Mvp-v, is defined as a binary relation on VP×V.
- For any a variation point vp and variant v such that vp ∈ VP and v ∈ V, if there exists a set such that {vp, v} ∈ Mvp-v, this indicates that v is included in a program only if vp is satisfied.



Domain knowledge:
Info. 1) *v0* was added to the program *B* because of the newly added feature *f0*.
Info. 2) *v0* was removed from the program *D* because of the deleted feature *f1*.
Info. 3) *v1* was removed from the program *D* because of the deleted feature *f2*.

Fig. 5. An Example of Code Change

We create VVMs based on the domain knowledge. For example, Fig 5 shows some code of program A, B, C, and D from Fig. 3. In the example, when program A and B are compared, the variant v0 is identified, and based on domain knowledge that "v0 was added to the program B because of the newly added feature f0", we can define {f0, v0} ∈ Mvp-v. Also, for another example, when we compared program C and D, the variant v1 is identified, and it is found that "v1 is not in program D because of the deleted feature f2" then we can define {f2, v1} ∈ Mvp-v.

2) Identifying mandatory features and their relationships is rather simple and straightforward. They can be identified from classes/methods that are common among the legacy programs.

3) In C++ language, class is defined as data structure using the keywords: class, struct, and union.

As steps 1 through 3 of the process (Fig. 4) are iteratively performed, VVMs are also continuously refined. For example, in Fig. 5, we have identified {f0, v0} ∈ Mvp-v before comparing program C and D. However, after comparing program C and D, we identify that v0 (that was previously added) was deleted. With the information of "v0 was added to the program B because of the newly added feature f0" and "v0 was removed from the program D because of the deleted feature f1" we identify that v0 exists when both f0 and f1 are included in products and modify VVMs accordingly; we refine {f0, v0} to {f0 ∧ f1, v0}.

The VVMs are used in the next step to identify the types of feature variability and relationships between variable features.

5.2 Identifying New Features and Feature Relationships

The purpose of this step is to identify the types of feature variability and feature relationships from VVMs defined in the previous step. The basic idea of identifying feature relationships is that if two variants have a program dependency (e.g., calls), then the features used in the specifications of the respective variation points may have relationships.

We first define three types of program dependency: composition, specialization, and operation dependency ; we classified the dependencies introduced in [22] into these three types based on structural and operational characteristics. These types of dependencies can be identified automatically from ASTs created in the previous step.

Definition 4. Composition relationship between variants:

- For any variants v1 and v2, we define a composition relationship, which means that v2 is contained in (i.e., is a member of) v1, if one of the following conditions is satisfied:
- v1 is a file and v2 is a class, method, variable, or statement defined or declared in v1.
- v1 is a class and v2 is a class, method or variable that is a member of v1.
- v1 is a method and v2 is a set of contiguous statements inside the body of v1.
- v1 is a statement and v2 is a variable used in v1.

Definition 5. Specialization relationship between variants:

- For any variants v1 and v2, we define a specialization relationship, which means that v1 is a specialization of v2, if one of the following conditions is satisfied:
- v1 and v2 are classes and v1 inherits v2.
- v1 is a method that implements a virtual method v2.

Definition 6. Operation dependency between variants:

- For any variants v1 and v2, we define a operation dependency, which means that v1 operationally depends on v2, if one of the following conditions is satisfied:
- v2 is a file and v1 is a statement that includes v2.
- v2 is a class and v1 is a variable that is an instance of v2.
- v2 is a method and v1 is a statement that calls v2.
- v2 is a variable and v1 is a statement that refers v2.

Then, we define the following seven heuristics for identifying variable features and feature relationships: 1) identifying optional features, 2) analyzing OR-expression on features, 3) analyzing XOR-expression on features, 4) analyzing AND-expression on features, 5) analyzing composition relationships, 6) analyzing specialization relationships, and 7) analyzing operation dependencies. In this paper, we describe these rules informally and briefly because of space limitation. Each rule is described below.

Rule 1. Identifying optional features: This is a basic rule for identifying optional features from VVM.

- If there exists a set such that {f, v} ∈ Mvp-v, we identify f as an optional feature.
- If there exists a set such that {¬ f, v} ∈ Mvp-v, we identify f as an optional feature.

Fig. 6 shows a code change between programs 1 and 2 (of the family of legacy Notepad++ programs) and VVM. By applying Rule 1, we define File Open History as an optional feature.

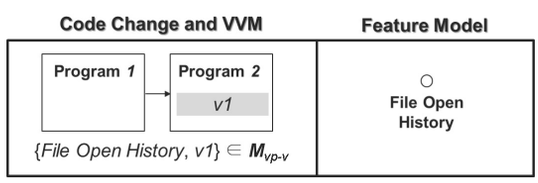


Fig. 6. An Example of Rule1

Rule 2. Analyzing OR-expression on features: Some variants may be mapped to OR (i.e., ∨) expression on features. OR-expression on features, e.g., f1 ∨ f2 (where f1 and f2 are features), is regarded as a new feature f3 representing the expression, i.e., f3 ≡ f1 ∨ f2 (We name f3 with domain experts). When we transform the expression to a feature model, we define f1 and f2 as optional features and also define a composed-of relationship between f3 and (f1, f2): f3 is composed of f1 and f2. (We apply a similar scheme to multiple-OR-expression on features, e.g., f1 ∨ f2 ∨ ⋯ ∨ fn, and regard it as a new feature representing the expression.)

Fig. 7 shows code changes between two Notepad++ programs and VVMs. By applying Rule 2, we define an optional feature XML-based Configuration as (GUIC ∨ XMLL ∨ FOH), which are all optional, and also define a composed-of relationship: XML Configuration is composed-of GUIC, XMLL and FOH (See the feature model in Fig. 7).

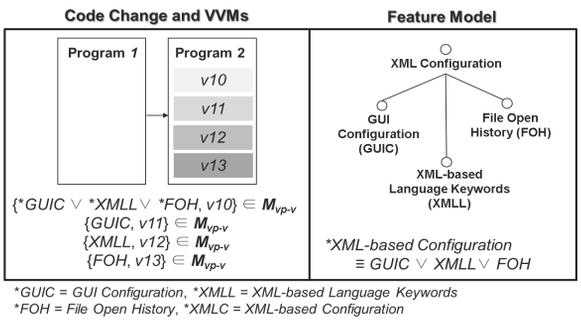


Fig. 7. An Example of Rule2

Rule 3. Analyzing XOR-expression on features: We analyze variants mapped to XOR (i.e., ⊕) expression on features in the similar way as Rule 2. XOR-expression on features, e.g., f1 ⊕ f2 (where f1 and f2 are features), is regarded as a new feature f3 representing the expression, i.e., f3 ≡ f1 ⊕ f2. When we transform the expression to a feature model, we define f1 and f2 as alternative features, and also define a composed-of relationship between f3 and (f1, f2): f3 is composed of f1 and f2. Fig. 8 shows an example of Rule 3. (We apply a similar scheme to multiple-XOR-expression on features.)

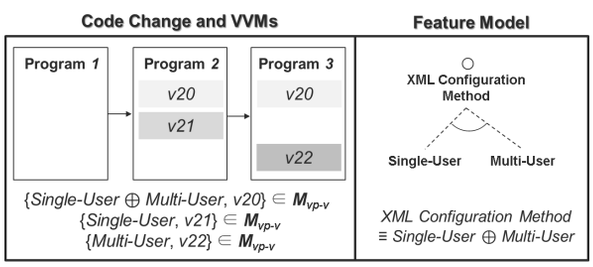


Fig. 8. An Example of Rule3

Rule 4. Analyzing AND-expression on features: Some variants may be mapped to AND (i.e., ∧) expression on features (e.g., f1 ∧ f2 where f1 and f2 are features). This case could be seen using a simple Venn diagram in Fig 9. In the Venn diagram, the overlapped part (2) indicates a set of variants mapped to (f1 ∧ f2). Based on (1), (2), and (3), we interpret (f1 ∧ f2) as follows (let (2) be a non-empty set):

- **Case 1)** If both (1) and (3) are empty sets, the feature f1 and f2 are always selected together. Thus, we define (f1 ∧ f2) as a new optional feature (e.g., f3) but do not define f1 or f2 as optional features.
- **Case 2)** Else, if only (1) is an empty set, f1 alone cannot be selected without f2. Thus, we define both f1 and f2 as optional features and define a require dependency: f1 requires f2.
- **Case 3)** Else, if only (3) is an empty set, f2 cannot exist without f1. Thus, we define both f1 and f2 as optional features and define a require dependency: f2 requires f1.
- **Case 4)** Else, if both (1) and (3) are non-empty sets:
- **Case 4.1)** If both variants in (1) and (3) have any direct or indirect[4] program dependency (such as composition, specialization, or operation dependency) on the variant(s) in (2), this means that the features f1 and f2 are always selected together (because both f1 and f2 require the overlapped part). It is meaningless to define each of f1 and f2 as optional, thus we define an optional feature, e.g., f3, that is composed of f1 and f2.
- **Case 4.2)** Else, if any variant in (1) has direct or indirect dependency on the variant(s) in (2), but not vice versa, this indicates that (2) is required to implement f1. Therefore, we define f1 and f2 as optional, and also define a require dependency "f1 requires f2."
- **Case 4.3)** Else, if any variant in (3) has direct or indirect dependency on any variant(s) in (2), but not vice versa, this means that (2) is required to implement f2. Thus, we define f1 and f2 as optional, and also define a require dependency "f2 requires f1."
- **Case 4.4)** Else, (i.e., the variants in (1) and (3) do not have any dependency on (2) directly or indirectly), f1 and f2 do not have any configuration relationships. (Note that in this case (2) indicates that f1 interacts with f2 when both of them are included in a product, but this does not mean that they have configuration dependency.) We define f1 and f2 as independently configurable optional features.

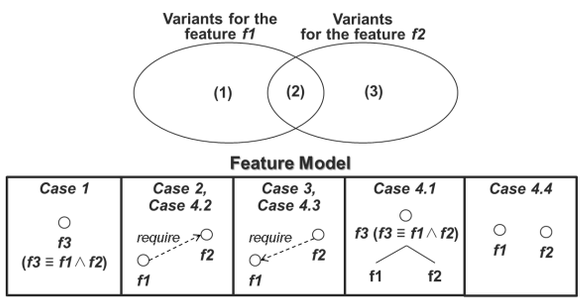


Fig. 9. An Abstraction of (f1 ∧ f2) and a Feature Model for Each Case

We apply a similar scheme to (f1 ∧ ¬ f2). In this case "f1 requires ¬ f2" is interpreted as "f1 excludes f2".

4) If A has a dependency on B, and B has a dependency on C, then we say that A has an indirect dependency on C.

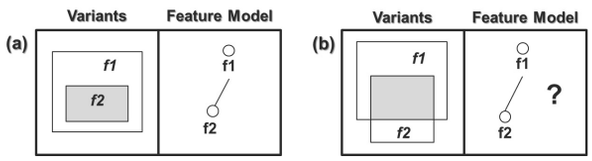


Fig. 10. An Example of Rule5

Rule 5. Analyzing composition relationships between variants: We may identify composed-of relationships between features from the composition relationships between variants. The Venn diagrams in (a) and (b) of Fig 10 show composition relationships between variants mapped to feature f1 and f2; the overlapped part indicates that the variant mapped to f2 is contained in the variant mapped to f1. If every variant mapped to f1 is directly or indirectly contained in any variant mapped to f2 ((a) of Fig. 10), we define a composed-of relationship: f1 is composed of f2. Otherwise ((b) of Fig. 10), it is difficult to decide whether a composed-of relationship exists between f1 and f2 automatically; we check if f2 is an optional feature and if there is a composed-of relationship between f1 and f2 with domain experts.

Rule 6. Analyzing specialization relationships between variants: We consider the specialization dependency between variants as a mechanism for implementing generalization-specialization relationships between features. (We call the generalization-specialization relationship as "gen-spec relationship" in short.)

- For any features f1 and f2, if any variant mapped to f1 directly/indirectly has specialization relationship with the variant mapped to f2, but not vice versa, we define a gen-spec relationship between f1 and f2.

Rule 7. Analyzing operation dependencies between variants: If a variant, e.g., v1, has a direct or indirect operation dependency on others, e.g., v2, this indicates that v1 requires v2 for correct behavior. Therefore, the operation dependency between variants imply a require relationship between features mapped to them.

- For any features f1 and f2, if any variant mapped to f1 directly/indirectly has an operation dependency on the variants mapped to f2, but not vice versa, we identify a require dependency: f1 requires f2.

- In the case where, features f1 and f2 are always selected together, we define an optional feature, e.g., f3, that is composed of f1 and f2 (We do not individually define f1 and f2 as optional features.).

- For any features f1 and f2, if any variant mapped to f1 has a direct/indirect operation dependency on the variants mapped to (¬ f2), but not vice versa, we identify a exclude dependency: f1 excludes f2.

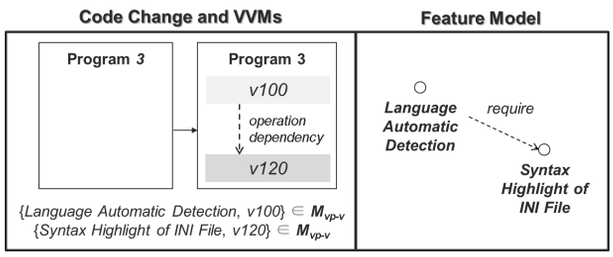


Fig. 11. An Example of Rule7

For example, in Fig. 11, a variant mapped to a feature Language Automatic Detection has an operation dependency on a variant mapped to Syntax Highlight of INI File. Applying Rule 7, we define a require dependency: Language Automatic Detection requires Syntax Highlight of INI File.

For VVMs that have complex code inclusion conditions that consist of multiple ∨, ⊕, ∧, and ¬ expressions on features, we can apply more than one rules. For example, if there exists a set such that {(f1∧ (f2 ∨ f3)), v} ∈ Mvp-v (where v is a variant and f1, f2, f3 are features), we can apply Rule 2 and Rule 4 in sequence.

In addition to the proposed rules, we may analyze features across the legacy programs to identify alternative features and/or exclude dependencies between features. For example, if only one from a set of features was included in legacy programs, the set of features could be alternatives.

The identified features and feature relationships are validated by domain experts in the next step.

5.3 Refining and Checking Integrity of Feature Model

As steps 1 through 3 (of the method process in Fig 4) are performed iteratively, we incorporate new features and feature relationships into the feature model, and check the logical integrity of the model. This step consists of two activities: 1) removing redundant relationships and 2) validating the features and feature relationships by domain experts.

Activity (1) Removing redundant relationships: Any relationships that can be deduced from others are removed.

- ***Redundant composed-of:*** For any two features f1 and f2, if the relationships "f1 is composed of f2", "f2 is composed of f3", and "f1 is composed of f3" are defined, we remove "f1 is composed of f3" ((a) in Fig 12 shows an example.). The same rule applies to gen-spec, require, and exclude relationships.

- ***Composed-of and require:*** For any two features f1 and f2, if we have "f1 is composed of f2" and "f2 requires f1", we remove "f2 requires f1." Composed-of implies require relationships between the composer and the composed features. (An example is shown in (b) in Fig. 12.)
- ***Gen/-spec and require:*** For any two features f1 and f2, if relationships "f1 is generalization of f2" and "f2 requires f1" were defined, we remove "f2 requires f1." Gen-spec implies require relationships between the generalization and the specialization features.
- ***Composed-of and Gen-spec:*** For any features f1 and f2, if we have "f1 is generalization of f2" and "f1 is composed of f2", we let domain experts to decide the correct relationship.

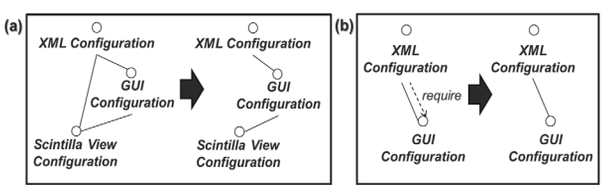


Fig. 12. An Example of Activity1

Activity (2) Validating the features and feature relationships: The features and feature relationships we identified are totally based on legacy implementations, thus we should check/confirm their integrity at the logical/conceptual level. The identified features and feature relationships need to be analyzed by domain experts. Any logically incorrect parts should be modified accordingly. Also, feature configuration dependencies that are necessary for any marketing reasons may be added by domain experts.

In the Notepad++ example, we identified optional features Language Automatic Detection and Syntax Highlight of INI File, and a require dependency: Language Automatic Detection requires Syntax Highlight of INI File (in Fig. 11) from the legacy programs. However, the dependency is logically incorrect; Language Automatic Detection is not limited to any specific language and can be selected/operated without Syntax Highlight of INI File. Therefore, we removed the dependency and made these features independently configurable.

5.4 Feature Modeling

After finishing steps 1 through 3 of the method process in Fig 4, we have a set of validated features and feature relationships, and these are "constraints" in feature modeling. In this step, we construct a feature model that satisfies these "constraints."

First, we define a feature graph. A feature graph is a labeled directed graph (F, Ec, Eg, Er) where F is a set of features identified in the previous steps, and Ec, Eg, Er $\subseteq$ (F $\times$ F) are directed edges such that if (f1, f2) $\in$ Ec, f1 is composed of f2; if (f1, f2) $\in$ Eg, f1 is generalization of f2; and if (f1, f2) $\in$ Er, f1 requires f2.

We developed the algorithm as follows:

- 1. Initialize a feature graph with a mandatory node "System." Add all variable features to F of the model. We now have a graph with disconnected nodes (without edges).
- 2. Add edges to the graph, based on feature relationships.
  - For composed-of, gen-spec, and require relationships between features, add edges connecting the features to Ec, Eg, and Er, respectively.
  - Add edges connecting the System node with the features that do not have parents, to Er (because System is composed of all features.)
  - Note that as composed-of implies require relationships between the composer and the composed features, if a require dependency such as "f1 requires f2" was defined, there is a possibility of a composed-of relationship "f2 is composed of f1." Therefore, we will select a subset of require edges and change them to composed-of edges to create a feature tree that consists of all Ec and Eg edges following the steps below.
- 3. For each node that do not have parent and that is connected by only one of edges in Er, move the edge from Er to Ec.
- 4. For each node that do not have a parent and that is connected by more than one edges in Er, select one of the edges and move it from Er to Ec. We need domain experts' opinion here. If feature description is available, we can use the method introduced by She et al. ([23]) to identify edges in Er that have high possibility to become composed-of edges. Continue this process until we find a sub-tree consisting of all edges in Ec and Eg. The resulting sub-tree is a feature model that satisfies the "constraints."
- 5. Validate the feature model with domain experts.

The left part of Fig. 13 shows the initial feature graph created after applying the algorithm steps 1 through 2, and the right part shows the final feature graph (i.e., the feature model for the asset) constructed after the algorithm step 4.

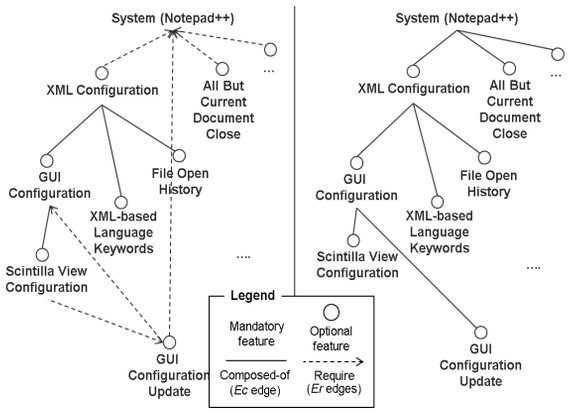


Fig. 13. Initial Feature Graph on the Left and the Final Feature Model on the Right

Now we have a feature model created through a variability analysis of the legacy products. This feature model is used in the next section to embed variation points to the SPL asset.

6. Feature-oriented Method: Feature-Oriented Product Line Reengineering

The purpose of this process is to engineer an asset consistent with the feature model (constructed in the previous process) in order to configure and manage the asset systematically based on the feature model. There are several techniques for inserting variation points to a program such as conditional compilation, aspect-orientation, etc. In our method, we use a conditional compilation technique to produce variant products using features as compilation parameters. The reasons we used this technique are: we can insert variation points to any code units of programs; it is a simple and widely used mechanism; and there exists various lexical preprocessors (e.g., C preprocessor [24], Antenna for Java ME [25], pure::variants [26], Gears [27]) supporting the mechanism. (Aspect-orientation can be an alternative if the programming language supports aspect-orientation.)

In this process we: 1) derive macro expressions from the variation point specifications; 2) reengineer the legacy programs to the product line asset with embedded variation points and variants using macro expressions; and 3) check consistency between the asset and the feature model. The steps 2 and 3 are performed iteratively and incrementally; if any inconsistency between the asset and the feature model is identified, the asset is reengineered to resolve the inconsistency. Details of these steps are discussed below.

6.1 Deriving Macro Expressions

To meet the variability requirements of the SPL, we must insert variation points into the asset code. We embed variants and variation points (found in the previous process) into the asset code using a conditional compilation technique. Activities for transforming variation point specifications to macro expressions using the feature model are as follows:

- First macro variables are derived from the names of variable features in the feature model.
- Based on VVMs, we define compilation conditions of the variants. For each variant, the corresponding variation point specification (identified from VVMs) is transformed to a conditional compilation condition that will be inserted with "#if" statement as a variation point to the code variant. The feature names included in the variation point specification are transformed to the corresponding macro names, $\wedge$, $\oplus$, $\vee$, $\neg$ operators used in the variation point specification are transformed to &&, ^, ||, and !, respectively. For example, if there exists a set such that $\{(f1 \wedge (\neg f2)), v\} \in$ Mvp-v (where v is a variant and f1 and f2 are features), we will insert "#if (f1 && (! f2))" to the variant v.

The identified variation points are embedded into the asset in the following step.

6.2 Reengineering the Legacy Programs to an Asset

The purpose of this step is to reengineer the legacy programs to an asset with embedded variation points and variants.

The first activity is to create a reference architecture that satisfies quality attributes of the SPL and is adaptable for products in the SPL. We can modify the legacy architecture, or create a new one if modification of the legacy architecture requires an excessive effort [28]. While modifying architecture, we have to carefully analyze interactions/dependencies between features, and architecture reengineering principles [28-29] could be applied. In the Notepad++ case study, we decided to reuse the legacy architecture of the latest version of Notepad++, because the architecture supports all of required features and satisfies the required quality attributes (e.g., performance).

Next, we modify the legacy components based on the architecture. Some components may be reused without modification, but others may need to be modified to satisfy functional/non-functional requirements. While modifying them, we continue to evaluate the quality attributes and any improvement actions (e.g., refactoring) can happen. (For

example, we removed code segments of unused features or duplicated code segments in legacy Notepad++ components.) Also, while modifying them, we updated VVMs to trace features to the reengineered asset. To meet the variability requirements of the SPL, the macro expressions identified in the previous step are inserted into component code.

To create an asset that are structurally and operationally consistent with the feature model, we continuously check the consistency between the asset and the feature model while reengineering the legacy programs; this activity is described in the next step.

6.3 Checking Consistency between the Feature Model and the Product Line Asset

The purpose of this step is to identify inconsistencies between the created asset and the feature model. If any inconsistencies are identified, we perform the previous step (described in section 6.2) again and reengineer the asset to resolve the inconsistencies.

The information used for consistency checking are VVMs and the feature model. The consistency rules are as follows:

- For any two features f1 and f2 of the feature model,
- 1) If f1 is composed of f2 or f1 is generalization of f2, there must be a composition or specialization relationship from the variant mapped to f1 to the variant mapped to f2 in the implementation.
- 2) If f1 is composed of f2 or f1 is generalization of f2, and if f2 is variable, the variant mapped to f1 must not have direct or indirect operation dependency with the variant mapped to f2.
- 3) If f1 does not require f2, f2 is not composed of f1 and f2 is not generalization of f1, the variant mapped to f1 must not have direct or indirect operation dependency with the variant mapped to f2.
- 4) If f1 excludes f2 or if f1 and f2 are alternatives, the variant mapped to f1 must not have direct or indirect composition, specialization, or operation dependency on the variant mapped to f2, and vice versa.
-

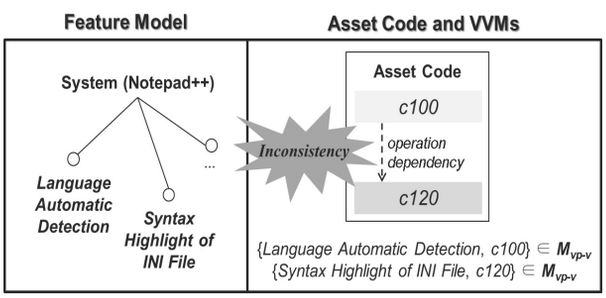


Fig. 14. An inconsistency example

For example, in Fig. 14, we do not have a require dependency between Language Automatic Detection and Syntax Highlight of INI File but, in the legacy implementation of Notepad++, the variant mapped to Language Automatic Detection has an operation dependency on the variant mapped to Syntax Highlight of INI File. Based on the 3rd consistency rule, we identified this inconsistency and reengineered the implementation.

The proposed method will be evaluated in the next section.

7. Evaluation

In order to evaluate our approach, we extracted an asset from the legacy products of Notepad++ and then compared the SPL version with the legacy products. From the legacy products of Notepad++, we selected five (from version 1.2 to 1.6) because the changes for these products have mostly been due to inclusion/exclusion of "major" features of source code editors, such as GUI Configuration, Multi-User based Configuration, Bookmark, etc. Using the method, we defined 24 optional features, and identified 30 feature relationships that are not described in documentation. The identified features and relationships are explicitly modeled in a feature model (Fig 13 shows a part of the feature model).

The feature model helped us understand the structure of the programs and analyze relationships between features hidden in the legacy code. When we validate the identified relationships, some relationships that are logically incorrect were modified (an example was explained in section 5.3). We derived macro expressions from the feature model, and reengineered the legacy programs while embedding the macro expressions. As we mentioned in section 6.2, we reused the architecture of the latest version (i.e., program 5). When we analyzed legacy programs, we could identify duplications across them. Some parts of the code that were used in the previous version were not used anymore (because corresponding features were excluded) but still existed in the following versions (e.g., 835 lines of code and 255 lines of comments in the program 5). While extracting the asset, such code was removed. Also, we improved the legacy design by encapsulating language defendant features and encoding standard features that have been changed frequently. Using the consistency rules, we identified some inconsistencies in the extracted asset, and reengineered the asset to resolve the inconsistencies (as shown in Fig 14).

When we analyzed the asset code, we found that variation points embedded in the asset were clearly mapped

to features of the feature model; the variation points were embedded as conditional compilation directives and macro names in the conditions are mapped to features of the feature model. Also, as the structure of the code is consistent with the structure of the features, it was easy to trace features to variation points embedded in the asset. To validate the asset, we generated legacy program from the SPL asset by selecting features specific to each legacy program, and checked whether the generated programs worked correctly. In addition, we instantiated new programs selecting new sets of features and tested the programs. As relationships between features were explicitly modeled, we could avoid selection of wrong combinations of features (e.g., selecting two features mutually exclusive). The asset code was integrated correctly according to the selection of features, and all instantiated programs worked correctly without any errors.

8. Related Work

As briefly mentioned in section 1, some papers presented methods for extracting an asset from a family of legacy programs developed by clone-and-own reuse. Bayer and others ([6, 7]) proposed a method for refactoring legacy components to an asset and used a decision model to capture variation points/variants of legacy software variants. It provides a good overview of which decisions exist, which files are influenced by the decision, and which decisions are used in a certain file. However, relationships between decisions were not explicitly capture and modeled (they were hidden in dependencies between files.). Alves and others ([8-10]) proposed a method for "bootstrapping" an asset from legacy systems and evolving the asset using aspect-oriented refactoring patterns. However, an aspect-orientation is a mechanism to insert variation points in implementation rather than model and manage relationships between aspects; it is difficult to understand the relationships between variation points. Alves et al. [8] mentioned that features could be mapped to aspects but they did not focus on separating a feature model from implementation, and maintaining the asset based on the feature model. As with other methods, relationships between variation points are mostly hidden in design/code and it is difficult to systematically extract and manage an asset. Unlike these methods, we explicitly model variation points and relationships between them in a feature model separate from an asset, and then extract and manage the asset based on the feature model.

As we construct a feature model bottom-up from a family of legacy programs, some researchers proposed methods to construct feature model bottom-up from legacy programs. Antkiewicz et al. [32] extracted features from a set of systems that extended the same framework. The purpose of their research is to extract a framework-specific model that represents instances of framework-provided concepts implemented in framework completion code. They used feature model as a framework-specific model and detected a set of patterns with source code queries to extract features. However, their work is limited to framework-based systems and requires a framework-specific modeling language [33] of a framework to extract features. Also, they cannot identify alternative features and configuration dependencies (i.e., require, exclude) between features. Our method can be applied to any family of legacy systems including framework-based systems.

Yang et al. [34] approach was based on detecting consistent data access semantics (i.e., similar usage of data entities by the methods in the applications) from similar open source systems that have similar data models. The records are then analyzed using Formal Concept Analysis to find the maximal set of objects (i.e., methods) that share a maximal set of attributes (i.e., data access semantics). Their method may be useful for data-transformation-based applications to identify business functions, but, with their method we cannot extract variable features that do not change data models, and also cannot identify configuration dependencies (i.e., require, exclude) between features.

Some researchers proposed methods to construct a variability model top-down from requirements documents. Rauf et al. [35] presented a framework allowing: 1) the specification of logical structures in terms of their content, textual rendering, and variability, and 2) the extraction of instances of such structures from rich-text documents. Their method is useful to identify structure of a variability model when requirements documents are available. However, structural/configuration relationships between features are usually hidden in the legacy programs rather than explicitly captured in documentation.

She et al. [23] proposed procedures for reverse engineering feature models from configuration files and feature descriptions (They also mentioned that the dependencies could be extracted from a code base but details were not presented.). In their approach, they assumed that variation points were embedded in a family of legacy programs. However, legacy programs developed by clone-and-own reuse mostly have been developed and maintained like single programs and usually do not embed variation points. In our approach, we defined procedures and rules for identifying variants/variation points of a family of

legacy programs that do not have embedded variation points, and analyzing relationships between the variation points. When constructing a feature model, She et al. identified structural relationships (i.e., composed-of, gen-spec) between features from feature descriptions. However, we identified the relationships from implementation because we wanted to identify all of feature relationships that are not explicitly captured in documentation.

Since the feature oriented domain analysis [18] was introduced, some researchers performed reengineering and refactoring from the feature-oriented perspective. Kang and others [28-29] reengineered a credit card authorization system based on a feature model to improve reusability of components, and reengineered legacy home service robot applications into an asset using a feature-oriented method [36]. Liu et al. [37] introduced the feature oriented refactoring (FOR) process which decomposes a system into features and reengineered the system based on the features. They reengineered an open source data base system implemented in Java using the FOR process. Trujillo et al. [38] re-engineered the AHEAD Tool Suite in a way similar to FOR. Although these researchers performed reengineering in terms of features, they did not consider extracting asset components from a family of legacy applications for use in creating an asset base for a product line. Their methods are not in the context of an extractive approach.

Satyananda et al. [39] provided a formal approach to verification of consistency between a feature model and component, as well as a connector view of the software architecture. They introduced a model for feature description and architecture description, and they illustrated a consistency verification approach that uses the Prototype Verification System (PVS). The difference is that their approach is based on a design description, while we rely on implementation.

Recently, there have been researches on extracting a variability model (e.g., feature model) from a family of legacy applications [40-42]. These researches can support legacy program analysis and feature modeling process of the proposed method. Also, there have been researches on checking consistency between a variability model and product line implementation [41, 43]. These researched can be used in the feature-oriented product line reengineering process of the proposed method.

9. Conclusion

Clone-and-own reuse based software development often has problems of degradation of software quality such as high maintenance costs, unused code in the program, and spaghetti/error-prone code. To overcome these difficulties, many organizations that have used clone-and-own reuse now want to migrate their legacy products to SPL. However, since most existing researches directly embedded variation points into the asset without modeling variation points separately, it was difficult to systematically engineer and maintain the asset. In this paper, we have established a method for creating a feature model from a family of legacy products developed by clone-and-own reuse and then engineering a feature-oriented asset that are consistent with the feature model.

We have shown how a feature model can play a key role in an extractive SPL approach. We applied the method to the family of legacy Notepad++ products and our analyses have demonstrate the feasibility of the method.

There are some issues related to the approach.

- **Detecting code not related to feature changes:** When we compare legacy programs using the AST based program comparison [21], we could detect code changes that are not related to feature changes, such as bug-fixes, naming convention changes, refactoring, etc. In the method, we analyze the code changes and then decide whether the changes are related to features; however, this may require intensive manual efforts if many changes are detected. To address this, we can use tools [44, 45] for detecting code changes caused by naming convention and refactoring but more researches are needed.
- **Tool support:** Every step of the processes (in Fig 4) is supported by tools. AST based program comparison tool [21] and reengineering supporting tools [30, 46] are available for steps 1 and 6, respectively. We are currently developing tools for supporting other steps as well.
- **Scalability of the method:** We have applied the method to Notepad++ products whose size is relatively small compared to other heavy commercial software (e.g., engine control systems), but we believe our method will scale up for product lines without dynamic variation. Stability of a product line can be an issue, not so much of the size.

This research is the first step towards a feature-oriented extractive approach to SPL. In this paper, we focused on creating a feature model from a family of legacy programs and embedding variable features (as variation points) into the asset code, in order to demonstrate that a feature model works well as the central model for variability management. In the next step, we plan to extend this method to support

feature-oriented product line architecture engineering based on our previous works [28-29], and also support dynamically configurable features.

References

[1] J. Bosch, "Design & Use of Software Architectures," Addison-Wesley, 2000.

[2] A. Maccari, "Experiences in assessing product family software architecture for evolution," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp.585-592. ACM, 2002.

[3] H. P. Jepsen et al., "Minimally invasive migration to software product lines," in *Proceedings of the 11th International Software Product Line Conference (SPLC)*, pp.203-211, 2007.

[4] K. Yoshimura et al., "Defining a strategy to introduce a software product line using existing embedded systems," *in Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, Seoul, Korea, Oct., 22-25, 2006.

[5] K. Yoshimura, "Model-based design of product line components in the automotive domain," *in Proceedings of the 12th SPLC*, Limerick, Ireland, Sep., 8-12, 2008.

[6] J. Bayer et al., "Transiting legacy assets to a product line architecture," in *Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, Sep., 1999.

[7] R. Kolb et al., "Refactoring a legacy component for reuse in software product lines: a case study," *J. Softw. Maint. Evol.: Res. Pract*, Vol.18, No.2, pp.109-132, 2006.

[8] V. Alves et al., "Extracting and evolving mobile games product lines," in *Proceedings of the 9th SPLC*, Rennes, France, Sep., 26-29, 2005.

[9] V. Alves et al., "Extracting and evolving code in product lines with aspect-oriented programming," *Transactions on AOSD IV (LNCS 4640)*, pp.117-142, 2007.

[10] V. Alves et al., "From conditional compilation to aspects: a case study in software product lines migration," in *Proceedings of the 1st Workshop on Aspect-oriented Product Line Engineering*, Portland, Oregon, USA, Oct., 22, 2006.

[11] D. Faust and C. Verhoef, "Software product line migration and deployment," *Softw. Pract. Exper*, Vol.33, No.10 pp.933-955, 2003.

[12] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in *Proceedings of the 24th ICSE*, Orlando, Florida, USA, May, 19-25, 2002.

[13] P. Frenzel et al., "Extending the reflexion method for consolidating software variants into product lines," in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*, Vancouver, BC, Oct., 28-31, 2007.

[14] H. P. Breivold et al., "Migrating industrial systems towards software product lines: experiences and observations through case studies," in *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, Parma, Italy, Sep., 3-5, 2008.

[15] M. V. Couto et al., "Extracting software product lines: a case study using conditional compilation," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, Mar., 1-4, 2011.

[16] Notepad++ [Internet], http://notepad-plus-plus.org/.

[17] Notepad++ Forum [Internet], http://sourceforge.net/projects/notepad-plus/forums/.

[18] K. C. Kang et al., "Feature-oriented domain analysis (FODA) feasibility study," Technical Report. CMU/SEI-90-TR-21, 1990.

[19] J. Lee et al., "A holistic approach to feature modeling for product line requirements engineering," *Requirements Engineering Journal (REJ)*, Sept., 2013.

[20] R. N. Charette, "This car runs on code," *IEEE Spectrum*, Feb., 2009.

[21] Eclipse C/C++ Development Tooling (CDT) [Internet], http://eclipse.org/cdt/.

[22] J.-L. Chen et al., "An object-oriented dependency graph for program slicing," in *Proceedings of the Technology of Object-Oriented Languages*, pp.121-130. IEEE, 1997.

[23] S. She et al., "Reverse engineering feature models," in *Proceedings of the 33rd ICSE*, Honolulu, HI, USA, May 21-28, 2011.

[24] International Organization for Standardization, "ISO/IEC 9899-1999: Programming Languages－C," 1999.

[25] E. Figueiredo et al., "Evolving software product lines with aspects: An empirical study on design stability," in *Proceedings of the 30th ICSE*, pp.261 - 270. ACM Press, 2008.

[26] D. Beuche, "Modeling and building software product lines with pure: variants," in *Proceedings of the 16th SPLC*, Vol.2, pp.255-255. ACM, 2012.

[27] BigLevel Software, Inc., Austin, TX, USA, "BigLever Software Gears: User's Guide," version 5.5.2 edition, 2008.

[28] K. C. Kang et al., "Re-engineering a credit card authorization system for maintainability and reusability of components - a case study," in *Proceedings of the 9th ICSR*, Torino, Italy, June, 11-15, 2006.

[29] K. C. Kang et al., "Feature-oriented re-engineering of legacy systems into product line assets - a case study," in *Proceedings of the 9th SPLC*, Rennes, France, Sept., 26-29, 2005.

[30] Understand - source code analysis and metrics [Internet], http://www.scitools.com/.

[31] J. Kerievsky, "Refactoring to patterns," Addison-Wesley, 2004.

[32] M. Antkiewicz et al., “Fast extraction of high-quality framework-specific models from application code,” *Autom. Softw. Eng.*, Vol.16, pp.101-144, 2009.
[33] M. Antkiewicz, “Framework-specific modeling languages,” PhD Thesis, Electrical and Computer Engineering, University of Waterloo, 2008.
[34] Y. Yang et al., “Domain feature model recovery from multiple applications using data access semantics and formal concept analysis,” in *Proceedings of the 16th WCRE*, Lille, France, Oct., 13-16, 2009).
[35] R. Rauf et al., “Logical structure extraction from software requirements documents,” in *Proceedings of the 19th International Requirements Engineering Conference*, pp.101-110, IEEE, 2011.
[36] K. C. Kang et al., “Feature oriented product line engineering,” *IEEE Software*, Vol.19, No.4, pp.58-65, 2002.
[37] J. Liu et al., “Feature oriented refactoring of legacy applications,” in *Proceedings of the 28th ICSE*, Shanghai, China, May, 20-28, 2006.
[38] S. Trujillo et al., “Feature refactoring a multi-representation program into a product line,” in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, Portland, Oregon, Oct., 22-26, 2006).
[39] T. K. Satyananda et al., “Identifying traceability between feature model and software architecture in software product line using formal concept analysis,” in *Proceedings of the International Conference on Computational Science and Its Applications*, Kuala Lumpur, Malaysia, Aug., 26-29, 2007.
[40] M. Acher et al., “On extracting feature models from product descriptions,” in *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, pp.45-54, 2012.
[41] D. M. Le et al., “Validating consistency between a feature model and its implementation,” in *Proceedings of the International Conference on Software Reuse*, pp.1-16, 2013.
[42] S. Nadi et al., “Mining configuration constraints: Static analyses and empirical results,” in *Proceedings of the 36th ICSE,* pp.140-151, 2014.
[43] A. R. Santos et al., “Strategies for consistency checking on software product lines: a mapping study,” in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, p.5, 2015.
[44] A. Loh and M. Kim, “LSdiff: a program differencing tool to identify systematic structural differences,” in *Proceedings of the 32nd ICSE*, Cape Town, South Africa, May, 2-8, 2010.
[45] M. Kim et al., “Ref-Finder: a refactoring reconstruction tool based on logic query templates,” *in Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Santa Fe, New Mexico, USA, Nov., 7-11, 2010.
[46] Lattix, Inc, Andover, MA, USA. “The LattixTM Approach - DSM for Software Architecture,” 2004.

Hyesun Lee

e-mail : hyesun.lee@etri.re.kr

She received a B.S. degree in Computer Science and Engineering (CSE) from Pohang Univ. of Science and Technology (POSTECH) in 2009. She received a Ph.D. degree in CSE from POSTECH in 2015. She has been a senior researcher at Electronics and Telecommunications Research Institute (ETRI) since 2015. Her research interests are in the area of software reuse, software product line engineering, and Internet of Things (IoT) platforms and systems.

Kang Bok Lee

e-mail : kblee@etri.re.kr

He received a B.S. degree in Electronic Engineering from Kyungpook National Univ. in 1993. He received a M.S. degree in Information and Communication Engineering from Chungbuk National Univ. in 2000. He completed a Ph.D. course in Information and Communication Engineering from Chungbuk National Univ. in 2002. He was a senior researcher at LG Semicon Co., Ltd. from 1993 to 2000. He has been a Principal researcher at ETRI since 2000. His research interests are in the area of RFID/NFC, ROIC, Biosignal processing, and IoT sensor application technology.