Recovering Object-Oriented Framework for Software Product Line Reengineering

Yijian Wu, Yiming Yang, Xin Peng, Cheng Qiu, Wenyun Zhao

School of Computer Science, Fudan University, Shanghai 201203, China
{wuyijian, 051021056, pengxin, 10212010021, wyzhao}@fudan.edu.cn

Abstract. A large number of software product lines (SPL) in practice are not constructed from scratch, but reengineered from legacy variant products. In order to transfer legacy products to SPL core assets, reverse variability analysis should be involved to find commonality and differences among variant artifacts. In this paper we concentrate on the recovery of SPL framework which can be represented by an object-oriented design model with variation points. We propose a semi-automatic SPL framework recovery approach with the assumption that involved legacy products have similar designs and implementations. In this approach, we adopt a bottom-up process based on clone detection and context analysis to identify corresponding mappings among design elements in different products. Then we use a top-down process from class level to method level with some heuristic rules to determine the commonality/variability classification and the variability type for each design element. In order to evaluate the effectiveness of our approach, we conduct a case study on an industrial product line and present comprehensive analysis and discussions on the results.

1 Introduction

Software Product Line (SPL) has been recognized as an emerging and effective paradigm for domain-specific software development with remarkable improvements on productivity, time to market and quality. In real-world SPL practice, development of a software product line rarely starts from scratch as product line engineering requires sophisticated domain experience [1]. A more popular situation for SPL adoption is that a company has already several variant products successfully developed in the domain, usually by ad-hoc copy-paste code reuse. In order to reduce risk of SPL adoption, the company usually chooses to reengineer those legacy variant products into a product line rather than to build one from scratch.

Reengineering for SPL transferring usually involves a series of similar legacy products in the same domain. Therefore, SPL reengineering should involve differencing and variability identifying for artifacts from different legacy products besides extracting them from source code, documents or execution traces. Although there have been some research on SPL reengineering and case studies [2-7], automatic or semi-automatic approaches for artifact differencing and variability analysis are seldom considered.

What we concentrate on in this paper is the recovery of SPL framework from legacy products. A framework is a reusable, "semi-complete'' application that can be specialized to produce custom applications, and usually targeted for particular business units and application domains [8]. By SPL framework, we mean an object-oriented design model with variation points and extension points for application product customization. For the purpose of SPL transferring, we try to recover an SPL framework from multiple similar legacy products by recovering a common design model (e.g. in UML class diagram) with variability. To that end, the following research problems must be addressed: how to identify the corresponding mappings of the design elements (e.g. classes, methods) among different legacy products; how to determine a design element to be common or different; how to further evaluate the variability type (optional, alternative or extensible) for each difference.

In this paper, we propose a semi-automatic SPL framework recovery approach which includes two stages: 1) a *mapping stage*, in which corresponding mappings among design elements from different products are established automatically; and 2) a *variability evaluation stage*, in which the variability type for each design element is identified. In the mapping stage, we adopt a bottom-up process based on clone detection and context analysis, first on method level and then class level. In the variability evaluation stage, we employ a top-down process from class level to method level, and use some heuristic rules to determine the commonality/variability classification and variability types for design elements. The recovered SPL framework is ultimately represented by an extended class diagram supporting variation point representation.

In order to evaluate the proposed approach, we conducted an experimental study on an industrial product line **DirectBank**, which had several variant products developed in different periods before they were reengineered into a product line. Our experiment has confirmed the feasibility of semi-automatic SPL framework recovery from multiple similar variant products by clone detection and context analysis. We also evaluate the effectiveness of our approach from two aspects: precision and recall of reverse variability analysis; the significance of the recovered SPL framework for further SPL understanding and transferring.

The remainder of this paper is organized as follows. Section 2 presents an overview of our approach, including the background, rationale and process. Section 3 and Section 4 describe the two stages of our approach respectively, i.e. mapping corresponding design elements among different variant products and evaluating variability types for all the mapped or unmapped elements. Section 5 evaluates the approach with an experimental study and presents some discussion. Finally, Section 6 discusses related work before Section 7 draws our conclusions.

2 Overview

In this section, we start with a general picture of our reverse variability analysis, showing the rationale behind. We then introduce the main process of our approach.

2.1 Reverse Variability Analysis

In forward SPL engineering, core assets with variations are created to support product-specific configuration and customization. In SPL reengineering, however, we expect to recover these core assets (also variability) from existing similar legacy products using a reverse engineering process.

A general picture of our reverse variability analysis is shown in Figure 1. The input is a set of legacy artifacts (including models) extracted from different variant products. The output is an object-oriented framework with abstract variations. The two stages (i.e. establishing element mappings and deciding types of variability) are explicitly labeled. The mapping stage considers the correspondence between elements. The mappings usually can be computed based on the similarity (either literally or structurally) between design elements. Specifically, two mapped elements do not necessarily have the same implementation, but possibly a very similar topological position in the design model. The variability evaluation stage is usually conducted according to the mapping results. Intuitively, for example, two mapped elements with completely different implementation are possibly *variant* elements.

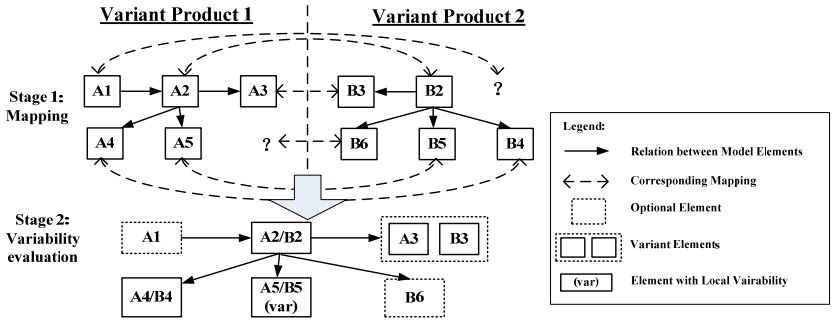

Fig. 1. A general picture of reverse variability analysis

In this paper, our ultimate goal is to recover an object-oriented design framework, especially a static class-based framework with variation points. A variation point here means a group of design elements that can be customized in application engineering, e.g. an optional element to be bound, an alternative element to be replaced by one of its variants. Based on our observations, we identify five variability types of design elements (see Table 1).

Table 1. Element variability in legacy products

Element variability	Description
Identical	mapped design elements that are exactly the same in different variant products
Variant	mapped design elements that have different implementations
Similar	mapped design elements that have some slight internal differences
Optional	a non-mapped design element that exists in some of the variant products
Product-specific	a non-mapped design element that exists in only one legacy product

The first three types (identical, variant and similar) are for *mapped* design elements which can be mapped with corresponding design elements in other variant products. The last two types (optional and product-specific) are for *non-mapped* design elements which cannot be mapped in all variant products. An optional element is not necessary for all products, while a product-specific element exists in only one legacy

product. These different types indicate different variability intent in the recovered SPL framework. For example, identical design elements usually belong to a common part of the framework; similar elements are usually common designs with local variations; and variant elements imply alternatives in the framework.

2.2 Recovery Process

The recovery process of our approach is presented in Figure 2. In our approach, each round of analysis takes two legacy variant products as inputs and tries to align design elements to recover an SPL framework with variability. More variant products can be incrementally compared and merged into the SPL framework.

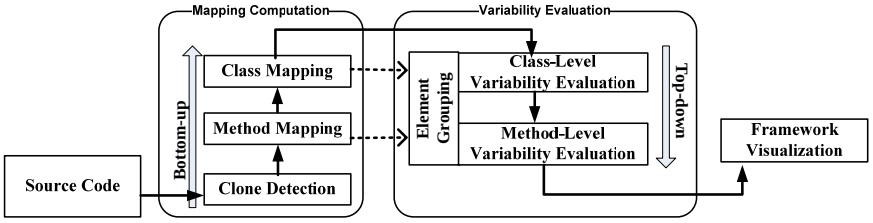

Fig. 2. An overview of our SPL framework recovery process

The major stages of our approach include mapping computation, variability evaluation, and an additional visualization phase. Mapping computation tries to align corresponding design elements for different legacy products. The mapping process follows a bottom-up process that conducts method-level mapping first and then class-level mapping (see Mapping Computation part in Figure 2). On method level, initial mappings are detected by clone analysis, and then internal similarity and external (context) similarity are considered to iteratively discover more mapping pairs. Similarly, class-level mapping first uses method-level mappings to directly establish some mapping pairs, and then employs design context to identify more mapping pairs.

After mapping computation, variability evaluation is conducted with a top-down process from class-level to method-level (see Variability Evaluation part in Figure 2). In class-level variability evaluation, each non-mapped class is evaluated to be optional or product-specific, and each mapped class is evaluated to be identical, similar or variant. For each similar class, method-level variability evaluation is conducted to determine more detailed variability similarly. The element variability decision result is then used for framework variability decision with design element groups.

Finally the recovered SPL framework is represented by framework visualization, using variability-extended UML class diagram.

3 Mapping Computation

In this section, we first describe the rationale of our mapping computation process with an illustrating example. Then we define a combined similarity measurement for

similarity computation, and introduce the sub-processes of method and class mapping respectively.

3.1 Rationale

An illustrating example of mapping computation between two variant products is shown in Figure 3. Intuitively, corresponding mappings can be identified by computing the similarity between two elements in a candidate pair. Some **internal similarity** measurements can be computed directly on a candidate pair. For example, elements Payment in both products (in Figure 3) are corresponding because they have the same name; elements Auditing and Audit are also very likely corresponding because their names are similar and (if we investigate their source code and find that) their source codes are similar. Besides, we also need some **external similarity** measurements to reflect similar roles that corresponding elements play in the design model. Usually, this external similarity can be measured by their structural contexts.

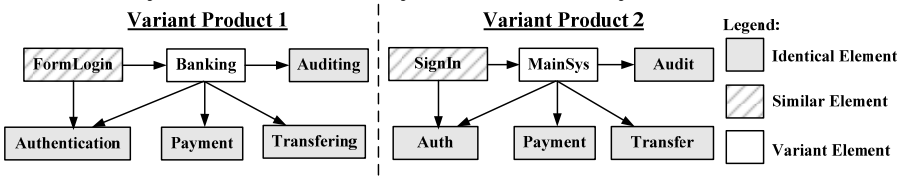

Fig. 3. An illustrating example of mapping computation

In the example in Figure 3, elements Banking and MainSys do not share name and (we assume that) their source code are not similar. Thus Banking and MainSys are not internally similar enough to be declared as a mapping. We then consider their context and find most of the context elements are *mapped*, which may lead us to believe that they actually form up a mapping pair.

It should be noted that structural contexts for mapping are based on other element pairs previously mapped. For example, if Banking and MainSys are mapped first, they can provide usable contexts when considering the mapping between FormLogin and SignIn, and vice versa.

3.2 Similarity measurement

In this subsection, we formally define internal, external similarity and the combined overall similarity between two design elements from different variant products.

3.2.1 Internal similarity

Currently, we consider name similarity and content similarity as internal similarity.

Name similarity measures the similarity between the names of two design elements (methods or classes). The name of an element includes the name of the package that the design element resides in. If the element is a class, we also consider

the class name. If the element is a method, the class name, the method name and the names of the parameters (if exist) are also included.

Given two design elements e_i, e_j, their name similarity is defined as the following:

$$SIM_{name}(e_i, e_j) = \sum_{type} w_{type} \frac{|W_i \cap W_j|}{(|W_i| + |W_j|)/2}, (0 < w < 1, \sum w = 1) \quad (1)$$

where W is the set of split words of the element e's full name and w is the weight of each type of name string. We believe that different parts of the name contribute differently to the name similarity. For example, the method names and the parameters play a more important role than package names when we decide whether two methods are similar or not.

Content similarity measures the commonality of the content of two design elements. On the method level, the content means the source code, and thus the content similarity can be measured by code clones. On the class level, we take methods as the content. Thus the content similarity for classes is measured based on the number of corresponding method pairs. Content similarity between two elements is generally defined as the following:

$$SIM_{content}(e_i, e_j) = \frac{|T_c|}{(|T_i| + |T_j|)/2} \quad (2)$$

where T_c is the collection of corresponding mapped contents within the scope of the element, and T_i and T_j are the collections of contents of element e_i and e_j, respectively. For methods, the contents are counted by the length of method (e.g. number of tokens); for classes, the contents are counted by the number of member methods.

3.2.2 External similarity

We now consider **context similarity**[9] as external similarity. Context similarity here presents structural similarity between two products. The context similarity of two elements is computed based on topological structure. For a method, the structure is the call graph, with a set of caller methods and another set of callee methods. We also consider the method signature and the class it resides in. All these elements are context of the method. For a class, the structure is a set of classes that have associations with the class under consideration.

We denote the context of an element e as $CT(e)$. To avoid considering all element pairs from $CT(e_i) \times CT(e_j)$, we consider only those pairs in the *DMS* (Determined Mapping Set, see 3.3 for detail) which involve elements in $CT(e_i)$ or $CT(e_j)$. We define Determined Context Mapping Set (*DCMS*) as a set of mapping pairs (e_{ci}, e_{cj}) where (e_{ci}, e_{cj}) $\in DMS$, $e_{ci} \in CT(e_i)$, and $e_{cj} \in CT(e_j)$. Then context similarity of two elements e_i and e_j is defined as the following:

$$SIM_{context}(e_i, e_j) = \begin{cases} \dfrac{|DCMS| * 2}{|CT(e_i)| + |CT(e_j)|}, & \text{if } |CT(e_i)| + |CT(e_j)| > 0 \\ 0, & \text{if } |CT(e_i)| + |CT(e_j)| = 0 \end{cases} \quad (3)$$

Particularly, if the context of both elements are not null and all elements in the context of both elements are in the DMS, the contextual similarity is 1; if neither element has a non-null context, the contextual similarity is 0.

In formula (3), a problem arise when the number of $|CT(e_i)|+|CT(e_j)|$ is small, meaning that the context of the element is not rich enough for analysis. Intuitively, we

believe that rich context will increase the *confidence* of the result of context similarity analysis. Otherwise, context similarity will contribute less than internal similarity. Therefore, we adopt the confidence as a weight balancing between external and internal similarity when calculating the combined similarity.

The **confidence** of context similarity is an additional, experimental value between 0 and 1 determined by the number of contextual elements. The richer the context is, the higher value is the confidence. Particularly, if no caller/callee methods exist (thus the contextual similarity is zero according to formula (3)), the confidence is not applicable; if the number of caller/callee methods is over a certain value (for example, five), the confidence can be one(1).

3.2.3 Combined similarity

Each of the above similarity measures a specific aspect of combined similarity. We believe that each aspect may contribute quite differently in different cases. Therefore, we define a combined similarity as the following:

$$SIM(e_i, e_j) = conf * SIM_{context}(e_i, e_j) + (1 - conf)(w_{name} SIM_{name}(e_i, e_j) + w_{content} SIM_{content}(e_i, e_j)) \quad \textbf{(4)}$$

where w_{name} and $w_{content}$ are weight for name and content similarity, respectively, $\sum w = 1$, and $conf$ is the confidence of context similarity.

In our approach, content similarity is covered by clone detection. Therefore, we simplify formula (4) into the following:

$$SIM(e_i, e_j) = conf * SIM_{context}(e_i, e_j) + (1 - conf) * SIM_{name}(e_i, e_j) \quad \textbf{(4')}$$

Formula (4') implies that, if confidence of context similarity is high, we may not take name similarity into account when deciding a mapping between two elements. Also, we will have to consider in the process to include the elements that are not discovered by clone detection.

3.3 Clone detection and method mapping

Theoretically, the mapping between any two methods can be decided by only calculating the values of combined similarities. But a practical problem is that calculating the context similarities of *all* pairs of methods from two products is difficult. Therefore, we perform clone detection before calculating name and context similarity to limit the initial set of method pairs.

Our method mapping process is started with source code clone detection. Methods are mapped based on the similarity value for each pair of methods. We define a **candidate mapping set** (CMS) to store possible corresponding method pairs and a **determined mapping set** (DMS) to store the mapping result. Basically, whether a pair of methods is determined as a mapping (i.e. added to the DMS) is based on the combined similarity value. The process is shown in Figure 4.

First, clone detection is applied to find method mappings across the (two) products under investigation based on ***content similarity***. We use a clone detection tool, Clone Miner, provided by Basit and Jarzabek, to find both simple clone class (SCC) and method clone class (MCC) [10]. The clone detection process provides a reduced result set of method pairs so that methods with the most similarity can be found first while

less similar methods are filtered out, reducing unnecessary analysis work. Clone instances of MCCs (cloned method pairs from different products) are taken as the initial CMS. With all clone instances, we will calculate name similarity and context similarity to find proper correspondences in the following steps.

The second step is to calculate ***name similarity*** for each method pair in the CMS. For each pair of methods in the CMS, if the name similarity value is above a given threshold, the method pair is moved to the DMS.

The third step is to add more method pairs to the DMS using ***context similarity***. For each method pair in the CMS, if the contextual similarity of a method pair exceeds a certain threshold, the method pair is moved to the DMS. Once a method pair is moved to the DMS, their context methods are paired up for internal similarity filtering; if their internal similarity is not very low, they are added to the CMS as a candidate pair. This operation is repeated until no method pairs are added to the DMS anymore.

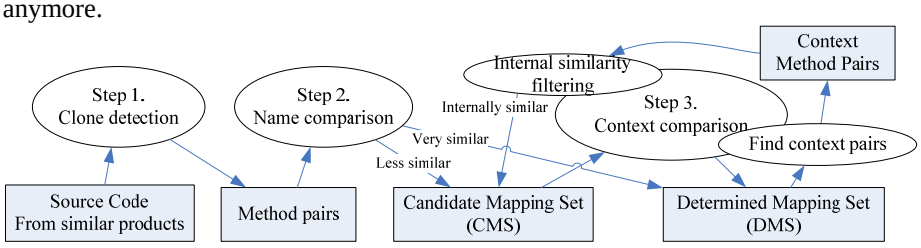

Fig. 4. The process for method mapping

The result mapping across products could be one-to-one, one-to-many or many-to-many. This cardinality will be used later when analyzing variability of the framework by **grouping** (see 4.2). In this step, however, all method pairs are just stored as-is.

3.4 Class mapping

Similarity between classes for establishing a mapping is calculated based on the similarities between the two classes. The ***name similarity*** of classes is calculated with the text string of the class' full names, similar to that of methods. The ***content similarity*** of two classes is defined by the number of corresponding mapped member methods and the total number of member methods. The implementation of each class (thus implementation of the methods) is not concerned. The ***context similarity*** of two classes is defined by the overlapping context of the two classes. The context of a class C includes three parts: the classes that C declares as a member, the classes that C declares in its member methods as a parameter type, return type or a local variable type, and the sub-/super-classes of C.

A class mapping is identified if the combined similarity of two classes is above a threshold. Class mapping pairs also have to be **grouped** for framework variability decision (see 4.2).

4 Variability evaluation

In this section, we present the top-down variability evaluation process. We first identify element variability for each element pair in the mappings with some heuristic rules, and then try to group elements in the mappings to decide framework variability.

4.1 Element variability decision

After element mapping, we have a collection of non-mapped elements and mapped element pairs. We now try to decide variability for each non-mapped element and for each mapped element pair. The decision process is shown in Figure 5, which is based on the mapping result and follows some heuristic rules. The variability decision process is top-down, from a class level to a detailed method level. It is also mentioned in Figure 5 that identical classes/methods and similar classes need to be further investigated for framework variability, as will be discussed in Subsection 4.2.

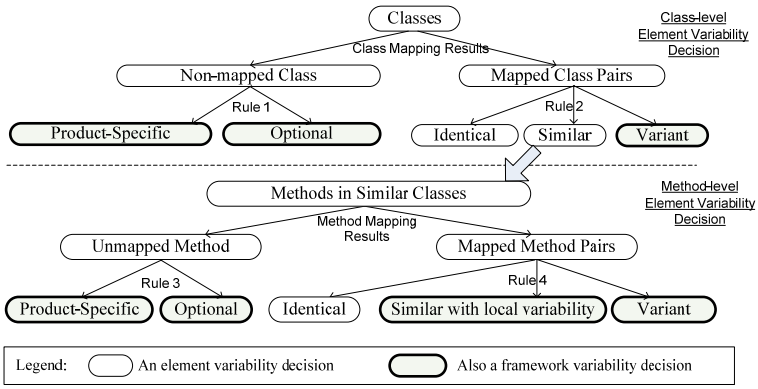

Fig. 5. Decision tree for element variability evaluation

In class-level variability decision, we check whether a class is mapped or not, according to class mapping computation. A non-mapped class can be either optional or product-specific according to Rule 1. For a mapped class pair, we evaluate the two classes in the pair to be identical, similar or variant according to Rule 2. As identical/variant classes represent exactly the same/different design elements, there is no need to further analyze method-level variability for them. For similar class pairs, further method-level evaluation is needed.

In method-level variability evaluation, we first check whether a method is mapped according to the results of mapping computation. Non-mapped methods are determined to be either optional or product-specific according to Rule 3. Mapped methods are evaluated to be identical, similar with local variability, or variant according to Rule 4.

Details of all the four heuristic rules are listed in Table 2. A difficult part of the rules is to distinguish product-specific and optional elements with Rules 1 and 3. In fact, we can only expect an approximate rule in most cases: product-specific elements only appear in one product, but optional elements usually appear in several but not all

products. Therefore, when considering only two variant products, we simply expect the appearance of an optional element in other potential variant products.

Rule 2 and Rule 4 are used to distinguish among identical, similar and variant elements based on content similarity. Identical elements have nearly the same content. Variant elements usually have quite different content, and they are mapped by their external (context) similarity. Similar elements are those between identical elements and variant elements, having considerable content and embody local differences at the same time. Therefore, Rule 2 and Rule 4 use two content similarity thresholds respectively to distinguish among identical, similar and variant elements.

Table 2. Heuristic rules for variability evaluation

Rule	Description
Rule 1: distinguish between **product-specific** and **optional** classes	If the corresponding classes of a class can be found in at least one other variant product, then it is **optional**; otherwise, it is **product-specific**.
Rule 2: distinguish among **identical**, **similar** and **variant** classes	If the content similarity among the corresponding classes is lower than $threshold_{c1}$, then the class is a **variant** class; if the content similarity is higher than $threshold_{c2}$ ($threshold_{c2} > threshold_{c1}$), then it is an **identical** class; otherwise it is a **similar** class.
Rule 3: distinguish between **product-specific** and **optional** methods	If the corresponding methods of a class can be found in at least one other variant product, then it is **optional**; otherwise, it is **product-specific**.
Rule 4: distinguish among **identical**, **similar** and **variant** methods	If the content similarity among the corresponding methods is lower than $threshold_{m1}$, then the method is a **variant** method; if the content similarity is higher than $threshold_{m2}$ ($threshold_{m2} > threshold_{m1}$), then it is an **identical** class; otherwise it is a **similar** class.

4.2 Framework variability decision

With the approach provided in the previous subsections, we have identified variability for design *elements* but there is still one step away from a *framework* with variability. Therefore, we further give an approach to decide the variation points in the framework based on ***element groups***. That is, we try to group a set of mapped elements to form up an element group as a variation point of the recovered framework.

An element group is a closure of all elements in a set of element mappings. For example, we have two element mappings (e_1, e_2) and (e_1, e_3), where e_1 is from product A and e_2 and e_3 are from product B. We then infer instinctively that it is possibly the case that e_1, e_2 and e_3 are all variants in the recovered framework at a variation point. Therefore, we group e_1, e_2 and e_3 into an element group {e_1, e_2, e_3} as a variation point. An element group could be a class group (CG) or a method group (MG).

In the recovered framework, each class group presents a class-level variation point and each class in the class group acts as a variant. Similarly, a method group presents a method-level variation point.

In Figure 6, we have determined element variability. Here are some simple rules for determine framework variability on class level.

Given a class group {Ci}, where Ci is a class from a certain legacy product. If any two classes in the class group are identical, the class group can be converted to a common class in the recovered framework. Otherwise, if at least one pair of classes in the class group is similar and other classes are identical, the class group can also be converted to a common class by accommodating some local differences. If at least

one pair of classes is variant, the class group should be converted to a variant class. In any cases, if a *null* element is contained in the class group (i.e. there is a class not mapped in a product), the framework class should be marked as optional.

For those variant classes, a detailed method level decision will be carried out. The method-level process is similar and will not be discussed in detail in this paper.

5 Evaluation and Discussion

5.1 The DirectBank project

We applied our approach in recovering a framework for two similar web-based legacy products in the **DirectBank** project, namely DBankV1 and DBankV2. They were developed for different customers, but the development of DBankV2 was based on DBankV1. The overall structure remained untouched, while developers made some modifications to DBankV1 to create DBankV2. The sizes of the two products are quite similar, as shown in Table 3.

Table 3. Size comparison of the two **DirectBank** products

Product	LOC	#Class	#Method	#Method (excluding getter and setter)
DBankV1	10615	42	437	168
DBankV2	10517	41	424	160

5.2 Experiment results

5.2.1 Mapping results

In the mapping stage, we found 965 method mappings. Among them, 32 were one-to-one method mappings with high content similarity, which were directly added to DMS; 105 were added to CMS in the first iteration. The rest of mappings were established by analyzing name similarity and context similarity.

For class mappings, only 2 classes in DBankV1 and 1 class in DBankV2 are not mapped to classes in the other product. DBankV1 provides a class named `DownFTPFile` for file transfer protocol (FTP) support and another class named `DBankUtils` for specific byte-level operations. We cannot find correspondences for these two classes in DBankV2. Meanwhile, DBankV2 has a class named `BOCWeb` to extend functionality of class `BOCDirectBank`, but DBankV1 does not have the class `BOCDirectBank`, thus `BocWeb` is not mapped to any classes in DBankV1.

All identified class-level mappings are meaningful. Only one potential mapping is not discovered: the mapping between class `framework.NcSockConnection` in DBankdV1 and class `framework.NetworkSockConnection` in DBankV2. The reason is that the implementations of the two classes are too different and the context is not rich enough. More evaluations will be discussed in Subsection 5.3.

5.2.2 Variability analysis and framework recovery

After corresponding mappings are established, we try to harmonize all these mappings into element groups.

In our case, 33 class groups are created. Among them, there are 30 groups that each contains only 2 classes (one from DBankV1, the other from DBankV2). This implies that these classes are very likely common classes in the recovered framework. Although there are possibly some differences between the two classes in the group, the differences can be easily eliminated. For example, differences in class `framework.DateUtils` in both products, which are caused by an upgrading, can be easily merged into one class.

The other 3 groups contain more correspondingly mapped classes. For example, a collection of business classes for bank payment (named after the name of the bank, such as `ICBCDirectBank`, `ABCDirectBank`, `BOCDirectBank`, etc.). Such classes usually present a variation point in the recovered framework, because different products may support one or more banks. The other two groups are `NetworkConnection` related classes and `PaymentInfoTransfer` related classes. Either group shows a variation point.

There are 248 method groups containing 763 methods (~3.1 methods per group on average). These method groups are later used for method-level variability decision for variant classes, which will not be discussed in detail in this paper due to page limit.

Based on the class/method groups, we successfully recovered a framework with variability. Figure 6 shows a partial framework at class level (a) with a zoomed-in view (b). The "CG 15" stands for Class Group 15 created by our automatic tool, which is a mapped class group consisting of banking payment classes in our experiment products. If more semantics are provided, the "CG 15" can be replaced by some meaningful name that helps a better understanding of the recovered framework.

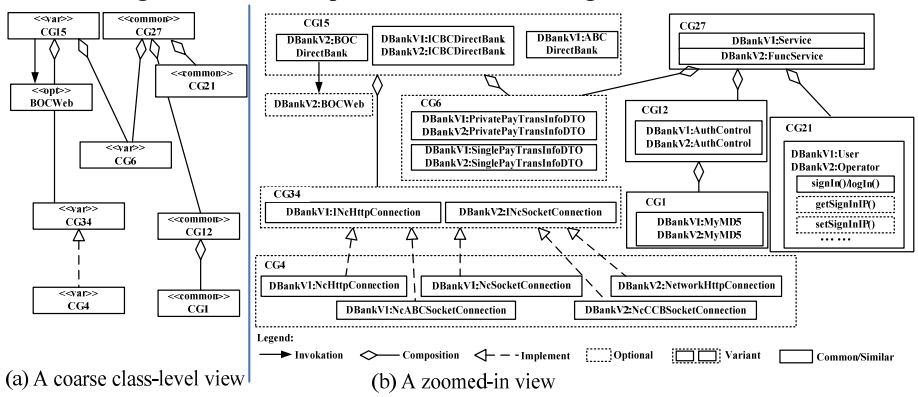

Fig. 6. Recovered (partial) framework at class-level

5.3 Evaluation

We have evaluated the precision and recall of the mappings at both class level and method level. The statistical result is shown in Table 4.

Table 4. Mapping results (class-level and method-level)

Mapping	All	All found	True-Pos	Precision	Recall
class-level	52	51	51	1.00(51/51)	0.98(51/52)
method-level	787	957	770	0.80(770/957)	0.98(770/787)

In Table 4, we can see that for class-level mappings both the precision and recall are quite high. For method-level mappings, we have checked all method mappings manually and find 770 method mappings out of 957 are meaningful, making a precision of 80%. The recall is a little bit difficult to calculate because the reference set of all potential method mappings is too large to identify manually. So we only checked all method pairs with high literal similarity (i.e. methods with similar names), and used those pairs confirmed by the developers as the reference mapping set. After analysis, we found only 17 method mapping pairs are not involved in the 770 true-positive mapping pairs (787 in total), making a recall of 98%.

After analysis, we find that an important contributor for the high precision and recall is the high similarity of our sample products. This implies that, if legacy variant products are architecturally similar enough, it is likely that our approach can recover precise mappings among classes and methods from different variant products.

On the other hand, we also confirmed that structural contexts help a lot in both class-level and method-level mapping computation. Intuitively, considering more structural contexts in mapping computation benefits the improvement of the precision and recall. Currently, we try to make more use of method context information (such as caller/callee methods) to identify potential mapping methods that have not so much direct similarity. In the result, we find 181 mappings established primarily due to high context similarity. These mappings were not detected by clone detection tool because they were not similarly implemented or their similar clone segments were too small to be detected. After applied the radical strategy, these contextually similar methods were finally mapped and added to the DMS, bringing great improvement of recall.

5.4 Discussion

5.4.1 Prerequisites and limitations to our approach

We adopt clone detection at the first step to process the source code to re-construct a common structure of the potential SPL. Thus, similarity of source code and design is necessary. Our approach will only apply to projects that satisfy the following prerequisites: 1) the legacy systems under investigation are developed in the same object-oriented language; 2) the legacy systems are similarly designed and coded, usually developed by the same team or person(s).

Different OO languages or incompatible architectural designs will require other approaches to identify mappings between systems under investigation. The framework recovery process is not able to identify the differences across various languages or design decisions.

Another limitation is the experiment settings. We tried several settings on Clone Segment Length, similarity thresholds, and the context confidence value. We find these values experimental and sometimes difficult to decide when applying our

approach in real SPL reengineering. Therefore, we suggest that the parameters of the approach be set according to the code style, domain type and product characteristics and be used only within an organization or development team.

5.4.2 How precise are the mappings?

In our experiment, we also find some flaws of our approach. There are some specious mappings of methods that are really difficult for us to decide what variability they carry in the recovered framework.

Methods with antonyms. There are several mappings containing methods with antonymous names, such as (DBankV1:`MessageCenter.getSuccessXML`(), DBankV2:`Message-Center.getErrorXML`()). These antonymous methods usually have similar names, implementation codes and invoking context. Usually, if such mappings exist, methods with the same name also form up mappings, such as (DBankV1:`Message-Center.getErrorXML`(),
DBankV2:`MessageCenter.getErrorXML`()). These two mapping cases will not be distinguishable unless more semantics are introduced. Fortunately, these method-level false mappings do not affect the mappings at class level. In our experiment, the classes named `MessageCenter` in both products are identical, and thus form up a common class in the recovered framework.

Getter & Setter methods and other small methods. There are more than 300 Getter&Setter methods in each of our experiment products. Usually these methods are too small to be detected by clone analysis due to clone detection settings. But as we begin to consider the Getter&Setters, we find they are quite useful for deciding class mappings. Also, there are many (> 400) small methods with the same name additionally added to our method mapping set when two classes are mapped, which contributes a lot in enriching the context.

Variability identification vs variability implementation. There are methods in a single product with very similar names. These methods, such as `BOCDirectBankpayToPrivate`(), `CommDirectBankpayTo-PublicSingle`(), `ICBCDirectBankpayToPrivateSingle`(), are easily mapped to each other for their high similarity in both name and context. Thus the variability is identified correctly. But as we look into these cases, we find that how to implement the variability in the recovered framework is arbitrary. We do not try to propose a guideline for refactoring existing implementation, but provide with a high-level object-oriented framework as the beginning of further engineering process.

5.4.3 Optional elements vs. Product-specific elements

In our experiment, we do not identify any product-specific element. The reason is that the designers believe most non-mapped elements are applicable to other (or future) products. The decision of whether a non-mapped element is optional or product-specific is experiential, as is also briefly discussed in Section 4.1.

Our sample products are comparatively small in size, and are similar enough. There are not many non-mapped elements, especially few non-mapped classes. Taking a few non-mapped elements in the recovered framework as optional is not a big threat for the complexity of the framework. In other cases where legacy products are not so similar that much more non-mapped elements are found, it would not be

feasible to include all these elements in the framework. To solve this problem, a product plan or an SPL scope plan is needed for the project managers or architects to decide whether to include a non-mapped element in the recovered SPL framework.

6 Related Work

Researchers and practitioners seek efficient approaches for integrating a family of legacy systems into a consistent architecture with controllable variations. SEI proposed Mining Architectures for Product line evaluation (MAP) [11] and Options Analysis for Reengineering (OAR) [12] for architecture recovery and variants identification. While a road map and practice guidelines are brought out, detailed techniques are not provided. Kolb [2] presents a case study in Ricoh company where a legacy component is to be refactored to accommodate Fraunhofer's PuLSE™-DSSA approach when creating a new product line from legacy products. The approach applied clone detection and variability analysis to refactor the component, rather than to recover a framework. Frenzel [4] provided a valuable extension to the reflexion model[13] to support software variants comparison on architecture level, but they did not "outline the technical details on how to reconstruct the architecture of the variants". John [5] focused on reusing legacy documents when establishing an SPL from legacy products. He argued a knowledge-rich approach for the process, but acquiring accurate and rich documentation is only too difficult. Knodel [6] presented a quality-driven approach to recover assets from existing systems and incorporate them in the SPL. His approach applied static, dynamic and historic analysis and was comparatively high-level, while ours is based on object-oriented source code to find commonality and variability of multiple products. Bianchi[14] proposed an iterative approach to reengineering legacy systems without interfering normal business operation and minimized the risk of refactoring. Although with different purposes from our recovery process, his process showed a feasible way to gradually reconstruct a new product line based on legacy systems. There are also some other reengineering work focusing on model recovery[15] and quality assurance[16].

The first stage (mapping stage) of our recovery process is closely related to research work in model differencing. Several differencing approaches and tools[17-19] could also be useful for establishing element mappings across products. In fact, any mechanisms that can efficiently establish corresponding mappings between design elements are applicable in our approach.

7 Conclusion and Future Work

Reengineering legacy products into software product lines is a practical choice for many companies to transfer their product development to SPL platforms. In this paper, we propose a semi-automatic method to support the recovery of SPL frameworks. The method is based on clone analysis and involves a two-stage recovery process, i.e. a bottom-up process for design element mapping and a top-down process for variability evaluation. In order to evaluate the effectiveness of our method, we

conduct a case study on an industrial product line and the results have confirmed the effectiveness of our method.

In our future work, we will try to extend our approach to efficiently handle more complex mappings in reverse variability analysis, e.g. legacy variant products with structural refactoring, to provide more practical techniques and tools for SPL reengineering.

Acknowledgments. The work presented is supported by National Natural Science Foundation of China (NSFC) under grants 60903013, 90818009 and 60703092.

References:

1. Pohl, K., Metzger, A.: Variability management in software product line engineering. In: ICSE 2006, 1049-1050. ACM (2006)
2. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: A case study in refactoring a legacy component for reuse in a product line. In: ICSM 2005, 369-378. IEEE (2005)
3. Lee, H., Choi, H., Kang, K.C., Kim, D., Lee, Z.: Experience report on using a domain model-based extractive approach to software product line asset development. In: ICSR 2009, 137-149. Springer (2009)
4. Frenzel, P., Koschke, R., Breu, A.P.J., Angstmann, K.: Extending the reflexion method for consolidating software variants into product lines. In: WCRE07, 160-169. IEEE (2007)
5. John, I.: Integrating Legacy Documentation Assets into a Product Line: In: 4th International Workshop on Software Product-Family Engineering, LNCS 2290, 78-101. Springer (2002)
6. Knodel, J., John, I., Ganesan, D., Pinzger, M., Usero, F., Arciniegas, J.L., Riva, C.: Asset recovery and their incorporation into product lines. In: WCRE05, 120-132. IEEE (2005)
7. Duszynski, S., Knodel, J., Naab, M., Hein, D., Schitter, C.: Variant comparison - A technique for visualizing software variants. In: WCRE08, 229-233. IEEE (2008)
8. Fayad, M.E., Schmidt, D.C.: Object-oriented application frameworks. Communications of the ACM, 40(10), 32-38 (1997)
9. Yang, Y. A Software Product Line Oriented Development Model and Reverse Eliciting Domain Components. Doctoral Dissertation. Fudan University (2010) (in Chinese)
10. Basit, H.A., Jarzabek, S.: A Data Mining Approach for Detecting Higher-Level Clones in Software. IEEE Transactions on Software Engineering, 35(4), 497-514 (2009)
11. Stoermer, C., O'Brien, L.: MAP - Mining Architectures for Product Line Evaluations. In: WICSA01, 35. IEEE (2001)
12. Smith, D.B., Brien, L.O., Bergey, J.: Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line. In: SPLC02, 316-327. Springer-Verlag, London (2002).
13. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE Trans. Softw. Eng., 27(4), 364-380 (2001)
14. Bianchi, A., Caivano, D., Marengo, V., Visaggio, G.: Iterative reengineering of legacy systems. IEEE Trans. Softw. Eng., 29(3), 225-241 (2003)
15. Nierstrasz, O., Kobel, M., Girba, T., Lanza, M., Bunke, H.: Example-driven reconstruction of software models. In: CSMR2007, 275-284. IEEE (2007)
16. Tahvildari, L.: Quality-driven object-oriented re-engineering framework. In: ICSM 2004, 479-483. IEEE (2004)
17. Collard, M.L.: An infrastructure to support meta-differencing and refactoring of source code, In: ASE 2003, 377- 380.IEEE (2003)
18. Maletic, J.I., Collard, M.L.: Supporting source code difference analysis. In: 20th IEEE International Conference on Software Maintenance, 210-219. IEEE (2004)
19. Canfora, G., Cerulo, L., Penta, M.D.: Ldiff: An enhanced line differencing tool. In: 31st International Conference on Software Engineering, 595-598. IEEE (2009)