A Systematic Methodology for Developing Component Frameworks*

Si Won Choi, Soo Ho Chang, and Soo Dong Kim

Department of Computer Science, Soongsil University
1-1 Sangdo-dong, Dongjak-Ku, Seoul, Korea 156-734
{swchoi,shchang}@otlab.ssu.ac.kr, sdkim@ssu.ac.kr

Abstract. Component-based software engineering (CBSE) is being accepted as an effective paradigm for building software systems with reusable components. Product line software engineering (PLSE) is an approach that utilizes CBSE principles to support the economic development of several applications in a domain. Hence, the components should conform to relevant domain standards and they must at least provide common functionality of the domain. Moreover, micro-level variability within commonality should also be modeled in components so that a product member-specific business logic or requirement can be supported through component tailoring or customization. Therefore, the degree of commonality and customizability determines the range of component applicability. In this paper, we propose a systematic approach to identify and model commonality and variability (C&V) and present a methodology to reason about the identified C&V model. With the proposed process and guidelines, components in a product line can better support a larger set of family applications.

1 Introduction

Component-based software engineering (CBSE) has been widely accepted as a new effective paradigm for building software systems with reusable components, consequently reducing efforts and shortening time-to-market. During the last decade, the industry practices of CBSE largely have been producing and utilizing in-house components in order to increase modularity and maintainability beyond object-oriented paradigm. A few case studies of CBSE have been focusing on producing domain common components.

Product line software engineering (PLSE) shares a great deal of CBSE conceptual elements and constructs, but its goal is to economically produce a family of applications by utilizing domain common components. Hence, it is an essential success factor in CBSE to model the common functionalities and features of a domain in order to produce such domain common components. Furthermore, the micro-level variability or alternatives within the commonality should also be modeled in such components so that a product member-specific business logic or requirement can be supported through component tailoring or customization [1]. Therefore, it is fair to state that the degree of commonality and customizability determines the applicability of components in PLSE [2].

* This work is supported by the research funding program of Soongsil University.

In this paper, we present a systematic methodology to identify and model the C&V. And, we show how the modeled C&V can be mapped to components and frameworks. In the appendix, we present a case study of building a banking system to show how the proposed methodology can be effectively and practically applied.

2 Related Works

The concept of Object-Oriented Application framework was introduced in order to increase reusability in [3]. Schmidt suggests "plug-in" which one of the different alternatives is plugged into a hot spot in a Framework [4]. Moreover, he proposes a hot spot subsystem as an implemented hot spot. In this study, abstract domain variability is embodied in a hook operation designed with polymorphism and inheritance. However, a larger reuse unit than objects such as component is not considered in this work.

COMO method by Lee [5] suggests a technique to extract common functionality into components by using a clustering algorithm. This method suggests identifying variation points and variants from a family requirement specification. FAST is an early product-line methodology which produces a process pattern for software production [6]. This pattern consists of three main processes; Qualify Domain, Engineer Domain, and Engineer Application. Especially the facility through domain engineering such as application engineering environment and application engineering process can be reused to produce family members rapidly.

Griss [7] suggests using feature model to derive the commonality and variability, where features are clustered into components. This work proposes a four-step process to use features to develop product lines. In addition, the issue of resolving crosscutting features is addressed. However, the process can be better augmented with specific work instructions, artifact templates and traceability framework. Kobra method by Atkinson [2] uses enterprise model, structural model, activity model, interaction model and decision model to model and to specify the variability for framework engineering. A stereotype «variants» is used in these models to indicate the existence of a variation point. A decision model is used to express the variation points for business processes and various diagrams. However, in this work, it is largely unspecified what criteria can be used to determine the existence of variability and how to identify variants and their scopes.

3 The Overall Process

The whole process to model C&V and design component framework consists of five phases and each phase has 2-3 activities as in figure 1. The process has a sequential task flow, but it can be applied iteratively. The details of each phase are specified in sections 3 through 7.

The first phase, *Requirement Normalization,* is to acquire a set of requirements from product members, to identify common vocabulary, and to re-write the requirements using the common vocabulary. This phase is essential to pursue subsequent phases since heterogeneity of different requirements is normalized so that requirements can be compared and C&V can be effectively applied.

The second phase, *Commonality Identification,* is to compare the set of normalized requirements and to identify the common features, i.e. functionality or quality attributes. Once a commonality set is identified, then a *Family Requirement Specification* is constructed from which components are modeled. Variability is a minor difference among family members in their logic or workflow, and it is realized into components so that component consumers can tailor acquired components for their own applications. The third phase, *Variability Identification,* is to identify variability and to design variation points.

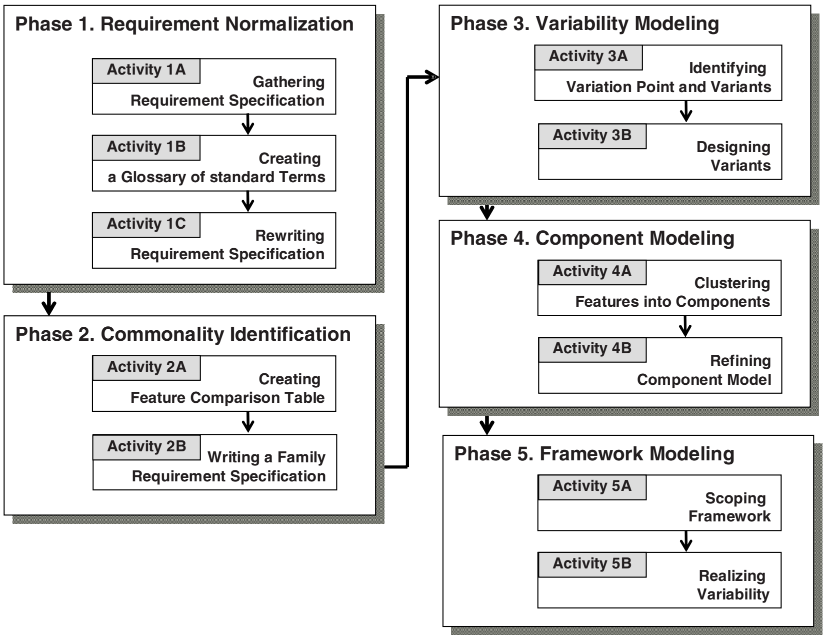
Phase 1. Requirement Normalization
Activity 1A
Gathering Requirement Specification
Activity 1B
Creating a Glossary of standard Terms
Activity 1C
Rewriting Requirement Specification
Phase 2. Commonality Identification
Activity 2A
Creating Feature Comparison Table
Activity 2B
Writing a Family Requirement Specification
Phase 3. Variability Modeling
Activity 3A
Identifying Variation Point and Variants
Activity 3B
Designing Variants
Phase 4. Component Modeling
Activity 4A
Clustering Features into Components
Activity 4B
Refining Component Model
Phase 5. Framework Modeling
Activity 5A
Scoping Framework
Activity 5B
Realizing Variability

Fig. 1. The Overall Process.

The fourth phase, *Component Modeling,* is to cluster features into components and to design preliminary components. Also, variation points are injected into these components and required interfaces are defined. A framework is a large-grained reuse unit which embodies a skeleton architecture, a set of related components and their relationships. An application is created by instantiating this framework. The last phase, *Framework Modeling*, is to identify related components and their relationships, which constitute a framework.

4 Requirement Normalization

This phase consists of three activities; Gathering Requirement, Creating a Glossary of Terms, and Rewriting Requirements.

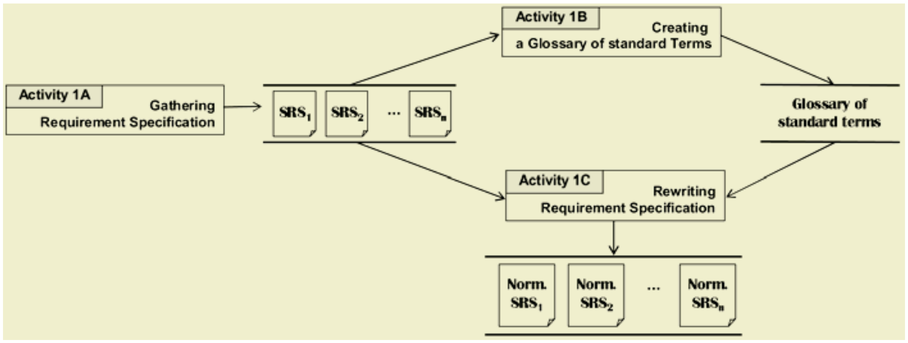
Activity 1A
Gathering Requirement Specification
SRS$_1$
SRS$_2$
...
SRS$_n$
Activity 1B
Creating a Glossary of standard Terms
Glossary of standard terms
Activity 1C
Rewriting Requirement Specification
Norm. SRS$_1$
Norm. SRS$_2$
...
Norm. SRS$_n$

Fig. 2. Requirement Normalization Process and Artifacts.

4.1 Gathering Requirement Specifications

The main goal of product line engineering is to develop reusable assets from which family members' applications can be instantiated in a cost-effective way. Hence, the domain analysis should be applied to a large set of product members, so that the developed components or framework can be widely reused. Activity *1A* is to collect a set of requirements from product members, and these are represented as SRS$_i$ in figure 2. Each requirement may come from a project member, or it can be constructed with a consideration of standard domain logic and knowledge.

4.2 Creating a Glossary of Standard Terms

One of the first obstacles in PLE is the heterogeneity of the requirement specifications gathered from several product members. The heterogeneity is largely appeared as inconsistency and ambiguity on terminology and concepts used in the requirements. A single term may have different meanings among product members, and several different terms among product members may have a single meaning.

Since the components used in PLSE should provide the standard or common features among product members, component producers must compare the set of requirements and identify a commonality set. But, this heterogeneity makes the comparison of various requirements impractical and inefficient. Hence, the set of requirements must be normalized using standard terms and concepts.

Activity *1B* is to derive a glossary of standard terms from the set of requirements gathered during activity 1A. In order to facilitate the process of identifying standard terms, we use a term comparison table as in table 1. We first grouping similar terms, T(1,1), T(2,1), … T(n,2) from the requirements and identify the most commonly used or standard term.

Table 1. Term Comparison Table.

Member / Category	M_1	M_2	···	M_n	Common Term
	T(1,1)	T(2,1)	···	T(n,2)	
	T(1,2)	T(2,3)	···	T(n,4)	

Once a term comparison table is constructed, we create a glossary of standard terms which is referred by all further activities and it provides a common definition of terms used.

4.3 Rewriting Requirement Specifications

By using the glossary of standard terms, we re-write the requirement specifications of product members. This will make it easier to compare the features among product members and to identify the commonality and variability since the revised requirement specifications will be expressed in all standard terms. However, if the requirement specifications from the product members are relatively homogeneous and there exists only minor difference in the terms used, then this activity can be omitted. In figure 2, a *Norm SRSi* is a re-written requirement specification, called normalized requirement specification.

5 Commonality Identification

During the first phase, requirement normalization, we normalized a set of heterogeneous requirements and domain knowledge. The second phase, commonality identification, will compare several requirements of product members to derive a set of common features among them. This common set will be used as the basis to determine the scope of candidate components and frameworks.

5.1 Creating a Feature Comparison Table

In order to effectively compare the set of requirements, we use a *Feature Comparison Table* as in table 2. For the 'n' members of the PL, their features are compared for potential commonness in this table.

Table 2. Feature Comparison Table.

Features	Product Members				Degree of Commonality	Rules Applied	Decision (Y/N)
	M_1	M_2	…	M_n			
F_1	√	√		√			
F_2		√		√			
…	√	√	√				
F_m	√		√	√			

A *feature*, as in PLE [7], is characterized by functionality and quality attributes. In many cases a feature maps to functionality. A PL has a set of features; F_1, F_2, .., F_n where F_i is a specific feature in PL. The first column lists all the features, F_1, F_2, ..., F_m, found in a product line. Earlier this set was defined as a union of the requirements.

A difficulty in creating this set in practice is to apply a consistent degree of granularity to all the features. This is because the granularity of features in a member's requirement may be different from that of other member's requirement. Hence, one should come up with an appropriate granularity level when there is a dispute on different granularity levels on a single feature. This granularity normalization can often be done with the intensive participation of domain experts.

Some of the features will be common among members while others are non-common, i.e. specific only to one or a few members. In the columns for product members, we express how each feature, F_i, is applied to each member, M_j. If F_i is applied to M_j, i.e. the member M_j requires F_i, a check mark is given. In deciding the applicability of features, we only consider the overall functionality and quality attributes at macro level.

In the column of 'Degree of Commonality', we specify a metric for each feature as (Number of check marks) / (Total Number of Members). This metric gives only an approximate degree of commonness for the given feature since one member's requirement may be more valuable or dominant than other members due to the different representation of the member in the domain.

In the column of 'Rules Applied', we specify the rules that have been applied in making decisions on whether or not each feature is included in the commonality set. In making decisions, we consider several factors; the degree of commonality, business influence of each member, sponsorship such as funding, and the degree of standardization. Although defining a set of precise rules for this decision making is not feasible, we propose the following candidate rules as a starting point;

i) If the degree of commonality is 100%, it is included in the set.
ii) If the degree of commonality is near 100% and there are some influential member s who require the feature, then it is included in the set.
iii) If the degree of commonality for a feature is relatively lower than those of other fe atures and there is no influential member who requires the feature, then it is not in cluded in the set. If a member is a key player in the domain in terms of business sc ale and market share or a client who pays the cost of developing reusable assets, th en it is included in the set.
iv) Other case which lies between the cases ii) and iii) should be judged with the dom ain knowledge, members' influence and business issues such as marketability.
v) If the feature is an essential intrinsic or standard functionality in a domain, then it can be included in the set with careful judgment.

The last column of 'decision' is about whether the feature should be included in the commonality set or not. The decision on whether or not a feature is common is mostly made based the above rules, but other business factors or domain constraints can also be considered.

5.2 Writing a Family Requirement Specification

Based on the comparison, we can now summarize the common features in *Commonality Specification Table* as in table 3. The first column is the identification number of each feature, and any reasonable number scheme can be used. The second column is for the names of features, and the third column is the description of features.

Table 3. Commonality Specification Table.

Feature ID	Feature Name	Description
CF_1		
CF_2		
. . .		

By using the common features in this table, we create a *Family Requirement Specification* which will be used in later phases as the reference for building components and frameworks. Also, the information in this table can be provided to component consumers so that they understand what services the components provide and what to expect from the reusable assets.

6 Variability Modeling

Through *Commonality Identification* activity, we have identified a common set of features that should be realized in components. However, a careful examination on a common feature often reveals a minor variation on logic or workflow. This is called variability within a commonality. By realizing this variability in developing components, the range of applicability and so reusability of components can be greatly increased [8].

6.1 Identifying Variation Point and Variants

A variation point of a feature in PLE is an identifier of a hot spot where the variability among different product members occurs [2]. In order to effectively model the variability, we use the *Variability Identification Table* as in table 4.

Table 4. Variability Identification Table.

Common Features	Variation type	Product Members					Range
		M_1	M_2	M_3	…	M_n	
CF_1	Logic	$V_{1.1}$	$V_{1.2}$	$V_{1.1}$		$V_{1.1}$	2
CF_2							
CF_3	Work-flow	$V_{3.1}$	Open	$V_{3.2}$		$V_{3.2}$	2+Open
. . .							
CF_m	Logic	Open	Open			Open	Open

In the second column, the type of a variation point is specified. In both CBD and PLE, a variation point can be in a form of logic and workflow in practice. The *logic* in this context consists of several steps which correspond to program statements once implemented. Hence, a variation point of *logic* describes a set of different algorithms for a system or business operation.

A variation point of *workflow* describes a set of different message flows for a system or business operation since a workflow is typically realized by a sequence of message invocations possibly over multiple objects or components. An example can be in a banking system product line that there can be different workflows, i.e. procedures, to evaluate and approve a loan application.

In the next set of columns, M_1, M_2, … , M_n, all possible instances of a logic or workflow variation point for CF_i, i.e. variants, are identified and specified. A variant, $V_{i,j}$ is j^{th} variant of i^{th} common feature, CF_i. In the first row for CF_1, the variant of M_1 is specified as $V_{1,1}$. If the variant for M_2 is not same as $V_{1,1}$, then M_2 is given a new variant ID. In the row, the M_3 is shown to have the same variant as M_1's. By repeating this procedure, all the possible variants for each variation points are identified.

A key problem in completing this table is to decide whether variability exists between any pair of product members. That is, what exactly is the difference between $V_{i,p}$ and $V_{i,q}$? We propose the following decision rules based on the elements of post condition, input domain, output range, and realization algorithms;

i) *If the post conditions of* CF_i *for two members are different, then there exists a vari ation.* A post condition is specified on the result of CF_i, and if the post conditions f or two members are different, it implies that the logics or workflows for two mem bers are different in some degree.

ii) *If input domains or output ranges of* CF_i *for two members are different, then there exists a variation.* An input domain for a feature is a set of all possible input value s and/or types, and an output range is the set of all possible values and/or types ge nerated by the feature. If input domains are different, then the logics or workflows to manipulate the input values will be different. Similarly, if output domains are di fferent, then there must be different algorithms or workflows to produce different s ets of output values.

iii) *If the realization algorithms for* CF_i *can be determined at this stage, and the algor ithms for two members are different, then there exists a variation.* In some cases, t he realization algorithms are not available until a later phase. But in some other ca ses, such algorithms can be available as a pre-fixed requirement. In this case, two algorithms can be compared to determine the existence of variability.

iv) *If none of the above rules can be applicable to* CF_i*, then other fact*ors such as pre-condition, invariants, and semantic description should be considered for a compari son.

The last column of *Range of Variation* specifies the total number of variants identified by decision rules. If this number is equal to 1, then there is no variability for the common feature. If this number is greater than 1, then CF_i has a variation point and a set of variants. If a variant for a variation point is unknown, i.e. open, then, it is marked with 'open'. If some variants are known and also some variants are open, then we put the total number of variants followed by the 'open'.

6.2 Designing Variants

From *Variability Identification Table*, we decide and specify how to realize the variants for each variation point. We use *Variability Range Table* as in table 5.

Table 5. Variability Range Table.

Variable Features	Type of Variation	Set of Variants	Open/ Closed	Default	Description
CF_1	Logic	$\{V_{1.1}, V_{1.2}\}$	Closed	$V_{1.1}$	
CF_3	Workflow	$\{V_{3.1}, V_{3.2}\}$	Open	$V_{3.2}$	
. . .					
CF_m	Logic	{ }	Open	None	

The first column lists only the common features that contain variation points, and so this information can be copied from the *Variability Identification Table*. The second column specifies the type of variation type which is also available in the *Variability Identification Table.* The third column, *List of Variants* lists all the variants identified. The next column, *Open or Closed*, specifies a binary value to indicate whether the list of variants identified is complete, i.e. closed, or expandable in the future, i.e. open. Depending on this openness, different implementation techniques can be adopted. The next column, *Default*, specifies a default variant among the list of variants, so that the components can be consumed without tailoring process if the default variant is needed during application engineering.

7 Component Modeling

7.1 Clustering Features into Components

During the previous activities, a set of common features and its variability scope have been identified. Based on this C&V model, conceptual components are designed. There is no mechanical procedure to identify components, but we apply the following guidelines to help clustering *related* features into components as in figure 3. Two features CF_i and CF_j are related if following conditions hold;

i) Features CF_i and CF_j are related if they belong to a same functional category as de fined by clients. This functional category may be based on system, module, functi onal classification and deployment classification. Typically, clients have in-depth domain knowledge and possess a functional classification scheme of features acco rding to their domain knowledge. And, so if two features belong to a same functio nal category defined by clients, then these are said to be related.

ii) Features CF_i and CF_j are related if they use common data or information. A featur e is a small-grained functionality required by clients, and each feature uses a set of data elements or information. If two features use exactly or mostly same set of dat a or information, they are grouped into a component.

iii) Features CF_i and CF_j are related if there is some strong degree of dependency bet ween the features. A dependency in OOP and CBD is a method invocation relation ship. Two features with dependencies should be clustered into a single component in order to minimize the coupling.
iv) Features CF_i and CF_j are related if they belong to the system layer and they proces s related system operations or transactions. Typically a feature that processes syste m operations belongs to a system layer, and the set of such features should be grou ped into a single component. Hence, these features can be distinguished from the f eatures that manipulate persistent data or objects.
v) Features CF_i and CF_j are related if they belong to the business layer and they mani pulate persistent data or objects. In contrast to iv), these features belong to busines s layer, and so they can be distinguished from the features that process system ope rations or transactions.

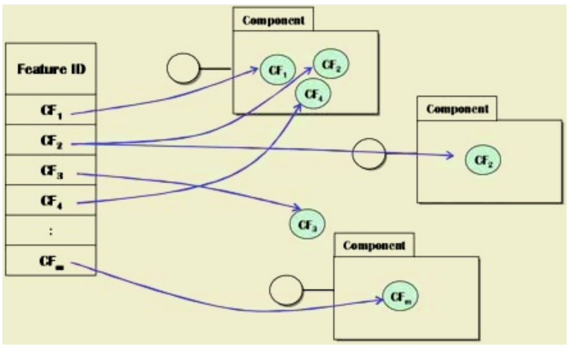

Fig. 3. Grouping Features into Components.

A feature is a system behavior exposed to clients, and so it tends map to a system component. However, a feature can map to a business component if the nature of the feature is mostly CRUD data manipulation. Hence, it is common in practice that a feature maps to a system component, which in turn invokes operations of a business component yielding an inter-component dependency.

7.2 Refining Component Model

The above set of decision rules is neither definitive nor complete since this grouping process largely depends on the domain knowledge and there can be exceptions to the rules. Two problematic cases that could be generated by applying the decision rules are unassigned features such as CF_3 and features appearing in multiple components such as CF_2 as shown in figure 3.

An unassigned feature can be grouped into a component that is the closest to the feature. If there are a large number of unassigned features, then they are grouped into a *utility* component. If a feature appears in multiple components, then we use the following decision rules;

i) If the functional nature of the feature is mostly information retrieval rather than inf ormation update, then the feature is duplicated into multiple components for conve nience and efficiency.

ii) If the functional nature of the feature is mostly information update rather than retri eval, then assign the feature to one component which uses the feature most intensi vely. And, let other components access this component with the feature through int erface. In this way, we reduce data/state inconsistency problem with duplicated fea tures while providing ways to access the feature.
iii) If the feature has the characteristics of the case ii), but it is not feasible to find a co mponent which will contain the feature, then, group such features into a common component. And, let other components access this feature through the interface of this common component.

Figure 4 shows that the unassigned feature and figure appearing in two components are re-configured according to the decision rules.

8 Framework Modeling

8.1 Scoping Framework

Once the components are identified, then frameworks are designed by grouping related components. A framework is semi-completed application and hence its granularity is larger than components. Therefore, in most cases, we have a single framework for a product line but there can be multiple frameworks in some cases. We use the following guidelines in determine whether two components are *related*;
i) Two components are related if both components are required to constitute a sub-sy stem. This is because a framework embodies a skeleton architecture of sub-system or a whole system.
ii) Two components are related if both components map to structural elements of a st able and skeleton application architecture of the product line.
iii) Two components are related if there is a dependency or association relationship. In ter-component relationships should be captured within a framework since a frame work is highly cohesive large-grained reuse unit.

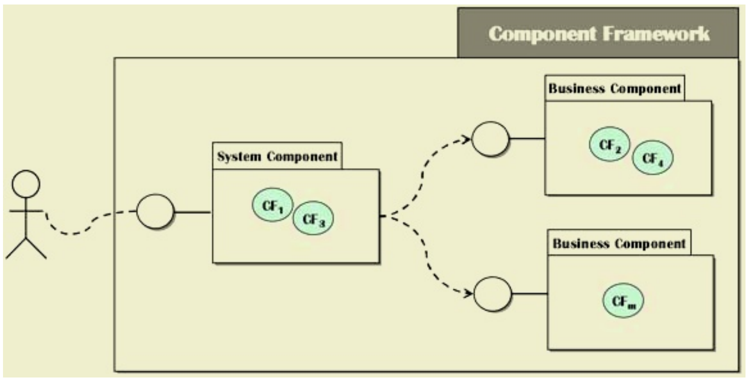

Fig. 4. Grouping Components into Framework.

Figure 4 shows that the three components are grouped into a single framework. The system component interacts with clients whereas business components act upon the invocation by system components through mediate pattern.

8.2 Realizing Variability

Once components are clustered into a framework, then we project the variability information specified in table 5 into frameworks. Typically variations points are realized inside components, and methods to set variants for variation points are defined in a *required interface* as shown in figure 5.

Tailoring components is different from invoking component methods in several ways. Tailoring components is typically done once per deployment or installation whereas invoking component methods are frequently made at run-time. The variant set during tailoring process will remain persistently within the component whereas actual parameters passed through method invocation are transient, i.e. short lived. Hence, a framework or components must maintain the variant set during tailoring process as persistent information. This is shown as *'CurrentVariant'* persistent attribute in figure 5.

If a variation point has a *Closed* scope, then it is tailored by using *Select( )* method. This method will take variants required by each application and store them persistently. If a variation point has a completely *Open* scope, then it is tailored by using *PlugIn( )* method. This method will take a reference to an external function, object or component, and invoke the method provided by the plugged in entity. If a variation point has a partially *Open* scope and some variants are known, then we use both *Select( )* method and *PlugIn( )* method.

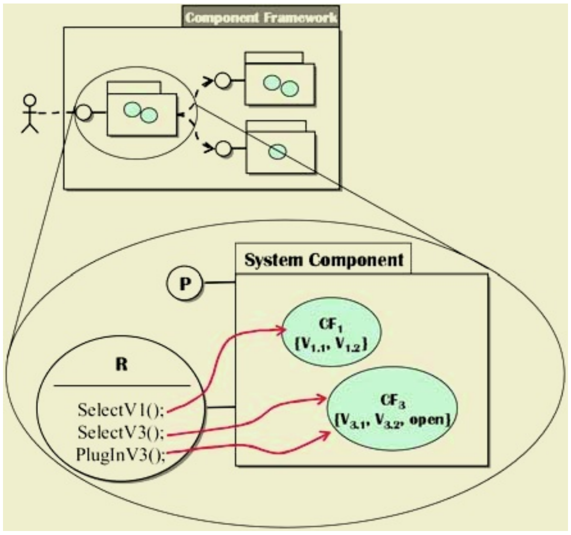

Fig. 5. Variation points projected into Framework.

As shown in figure 5, the CF_1 has a logic variation point of *Closed* scope and its tailoring method in the required interface is defined as *SelectV1();*. In the case of CF_3, the workflow variation point has an *Open* Scope with two known variants; $V_{3.1}$ and $V_{3.2}$. Therefore, two tailoring methods are used; *SelectV3()* and *PlugInV3()*. If one of the two known variants is required for a product member, then the *SelectV3( )* method is invoked. If the built-in variants, i.e. workflows, cannot be applied to the product member, then a plug-in object will be passed through *PlugIn( )* method.

9 Traceability

The process proposed in this paper includes 11 activities, and each activity produces one or two artifacts as summarized in figure 6. We now show the traceable items between pairs of artifacts in figure 8. The arrow between a pair of artifacts indicates the transformation direction, and the expression *Item1*→ *Item2* on arrows indicates the *Item1* is a source artifact from which a target *item2* is derived. And, so the target artifact can be traced to the source artifact through the transformation items. For *variants* in the artifact *Variability Range Table* are source items from which *variation design* and *required interface* in a *framework* are derived as in the figure.

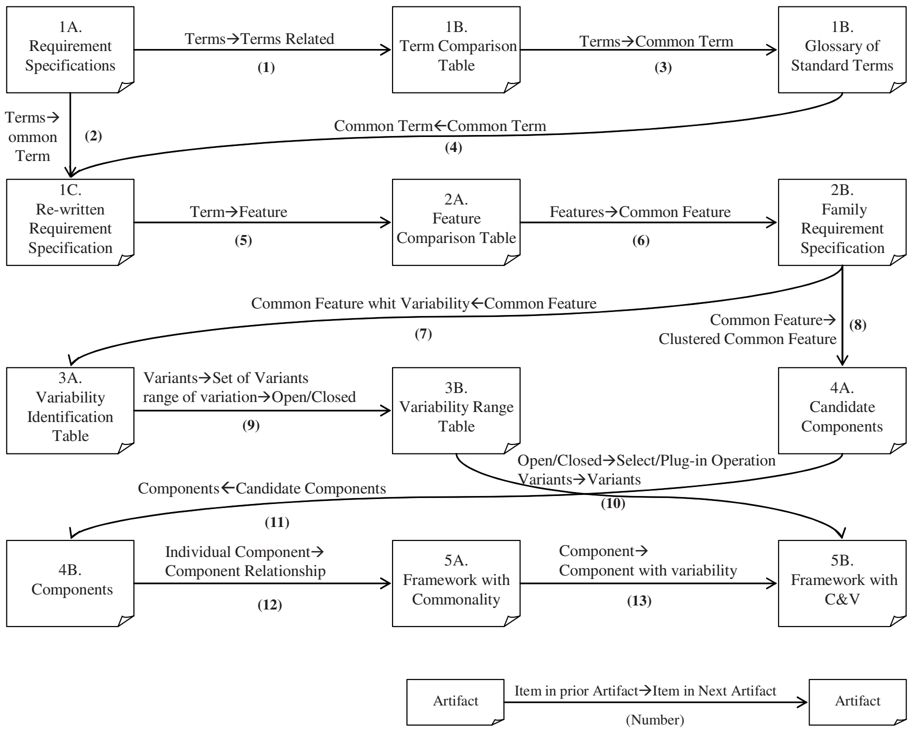
1A. Requirement Specifications
Terms→Terms Related
(1)
1B. Term Comparison Table
Terms→Common Term
(3)
1B. Glossary of Standard Terms
Terms→ ommon Term
(2)
Common Term←Common Term
(4)
1C. Re-written Requirement Specification
Term→Feature
(5)
2A. Feature Comparison Table
Features→Common Feature
(6)
2B. Family Requirement Specification
Common Feature whit Variability←Common Feature
(7)
Common Feature→ Clustered Common Feature
(8)
3A. Variability Identification Table
Variants→Set of Variants
range of variation→Open/Closed
(9)
3B. Variability Range Table
4A. Candidate Components
Open/Closed→Select/Plug-in Operation
Variants→Variants
(10)
Components←Candidate Components
(11)
4B. Components
Individual Component→ Component Relationship
(12)
5A. Framework with Commonality
Component→ Component with variability
(13)
5B. Framework with C&V
Artifact
Item in prior Artifact→Item in Next Artifact
(Number)
Artifact

Fig. 6. Traceable Items among Artifacts.

Each mapping in figure 6 specifies a transformation of a source item onto a target item, and a set of rules, called *traceability rules,* can be defined to show the validity of transformation. Due to the paper length, we only show rules for the mapping (10) Variability Range Table to Framework with C&V as an example.

- Rule 1. Each variant with Closed feature is mapped to a customization method Select<VP name> (VariantType v) in Required interface. The VariantType must be a datatype that specifies a set of all possible variants, so that an argument of VariantType can be passed by component clients.
- Rule 2. Each variant with Open feature is mapped to a customization method PlugIn<VP name> (PlugInObject p) in Required interface. The PlugInObject must be a class type that models a set of all variant objects that can be passed.
- Rule 3. A feature with both known variants and Open range must be mapped to two customization methods; a Select<VP name>( ) for known variants and a PlugIn<VP name>( ) for Open variants.

Similarly, set of traceability rules can be defined for other mappings.

10 Conclusion

Product line software engineering is a practical framework that utilizes CBSE principles in order to support the economic development of a set of applications in a domain. Hence, the components used in PLSE should conform to relevant domain standards or they must at least provide common functionality of a domain. Also, the variability should be modeled in components so that a product member-specific business logic or requirement can be supported through component tailoring or customization.

In this paper, we proposed a 5-phase process to identify and model the commonality and variability (C&V) and present a framework to reason about the identified C&V model in order to enable effective implementations of PLSE components. Activities within a phase are given a set of instructions and artifact templates. The whole process has been applied to a case study of banking domain. In addition, the traceability among artifacts and guidelines to enforce the traceability were given. With the proposed process and guidelines, the C&V can be systematically modeled into component framework, and the quality of delivered frameworks can be increased by applying traces using the proposed traceability foundation.

References

1. D'souza, D., *Objects, Components, and Frameworks with UML*, Addison Wesley, 1999.
2. Atkinson, C., et al., "Product Line Concepts", Chapter 14 of *Component-based Product Line Engineering with UML*, Addison Wesley, 2001.
3. Fayad, M. and Schmidt, D., "Introduction," *Communications of the ACM*, Oct. 1997.
4. Schmidt, H., "Systematic Framework Design by generalization," *Communications of the ACM*, Oct. 1997.
5. Lee, S., Yang, Y., Cho, E., and Kim, S., "COMO: A UML-Based Component Development Methodology", *Proceedings of Asia-Pacific Software Engineering Conference (APSEC99)*, Takamachu, Japan, pp. 54-61, Dec. 7-10, 1999.

6. Weiss, D. and Chi, T., *Software Product-Line Engineering: a Family-Base Software Development Process*, Addison Wesley, 1999.
7. Griss, M., “Product-Line Architectures,” Chapter 22 of *Component-Based Software Engineering*, Addison Wesley, 2001.
8. Kim, S., and Park, J., “C-QM: A Practical Quality Model for Evaluating COTS Components,” *Proceedings of International Association of Science and Technology for Development (IASTED) International Conference on Software Engineering*, Innsbruck, Austria, pp.991-996, Feb. 10-13, 2003.
9. Kim, S., “Lessons Learned from a Nationwide CBD Promotion Project,” *Communications of the ACM*, Oct. 2002.