Improving Product Line Architecture Design and Customization by Raising the Level of Variability Modeling

Jiayi Zhu[12], Xin Peng[12], Stan Jarzabek[3], Zhenchang Xing[3], Yinxing Xue[3], Wenyun Zhao[12]

[1] Shanghai Key Laboratory of Intelligent Information Processing
[2] School of Computer Science, Fudan University, Shanghai, China
[3] School of Computing, National University of Singapore, Singapore
072021130@fudan.edu.cn, pengxin@fudan.edu.cn, stan@comp.nus.edu.sg, xingzc@comp.nus.edu.sg, yinxing@comp.nus.edu.sg, wyzhao@fudan.edu.cn

Abstract. Product Line Architecture (PLA) plays a central role in software product line development. In order to support architecture-level variability modeling, most architecture description languages (ADLs) introduce architectural variation elements, such as optional component, connector and interface, which must be customized during product derivation. Variation elements are many, and design and customization of PLA at the level of individual variation elements are difficult and error-prone. We observed that developers usually perceive architecture variability from the perspective of variant features or variant design decisions that are mapped into groups of architecture variation elements. In the paper, we describe heuristics to identify configurations of variation elements that typically form such groups. We call them variation constructs. We developed an architecture variability management method and a tool that allow developers to work at the variation construct level rather than at the level of individual variation elements. We have applied and evaluated the proposed method in the development and maintenance of a medium-size financial product line. Our experience indicates that by raising variability modeling from variation element to construct level, architecture design and customizations become more intuitive. Not only does our method reduce the design and customization effort, but also better ensures consistent configuration of architectural variation elements, avoiding errors.

Keywords: software product line, architecture, variability, ADL

1 Introduction

Product Line Architecture (PLA) plays a central role in software product line development. A PLA differs from traditional software architecture for a single product in that it must be customizable for different products. Therefore, product line architecture description languages (ADLs), e.g. xADL 2.0 [1], introduced architectural variation elements such as optional component, connector and interface.

Designers usually perceive architecture variability from the perspective of

variant features or variant design decisions that are mapped into configurations of architecture variation elements. Figure 1 shows a typical design of the optional feature *Log* in a PLA. The architecture is depicted using boxes for components, small boxes with arrows for interfaces (inward arrows represent supplier interfaces; outward ones represent client interfaces), lines for links and ellipses for connectors. Dashed lines indicate optional elements. In order to model architectural representation of *Log*, an optional component *Log*, an optional interface in component *OnlinePayment*, two optional links and an optional connector have to be introduced. As discussed in Section 2, a simple composition between two components may induce 2^{11} combinations of basic variation elements. It is difficult and error-prone to model, customize and modify a PLA at the level of individual variation elements. For example, if we select the optional client interface in component *OnlinePayment*, we must also select relevant connector, links and component *Log*, or else the architecture after customization will not be valid.

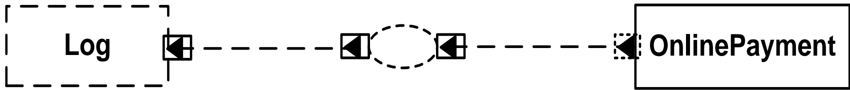

Fig. 1. A typical design of optional feature ***Log***

We observed that in general it is the case that certain groups of variation elements must be configured together in certain way to ensure correctness of the product architecture. We refer to valid group of variation elements that must be managed together as variation construct in the rest of the paper. In the design shown in Figure 1, optional feature ***Log*** is mapped into a single variation construct, but in general we have many-to-many mappings between features and variation constructs in PLA, which means a variant feature may involve several variation constructs. We hypothesized that it might be easier for designers to work with variation constructs instead of variation elements. We thought that variation constructs can simplify PLA design and customization, and make it less error-prone.

To test the hypothesis, we developed architecture variability modeling and customization method, and a tool that implements the method. In the method, we specify rules to help identify valid combinations of variation elements, mostly from the aspect of syntax. Then we further identify useful variation constructs according to some principles and clarify their different intention of variability design, as basic blocks for variability design and customization. We have successfully applied the proposed method in the development and maintenance of a medium-size financial product line. Our experience indicates that by raising the level of variability modeling and management not only do we reduce the design and customization effort, but also better ensure consistent configuration of architectural variation elements.

The remainder of this paper is organized as follows. Section 2 analyzes the problems in architectural-level variability design and customization after some background introduction. Section 3 defines PLA variation rules based on a PLA meta-model. Section 4 introduces our variation constructs with some examples. Section 5 presents our prototype tool implemented and evaluates our method with a case study on an enterprise product line. Finally, Section 6 discusses related work before Section 7 draws our conclusions.

2 Background and Problem Analysis

Software architecture reveals abstract views of the structure, behavior, and key properties of a software system [2]. In our work, we concentrate of structural architectural views that are typically expressed in terms of components, connectors, and constraints on the interactions among components. Different from single product architecture, product line architecture (PLA) defines a reference architecture shared by a family of products in a given business domain [3, 18, 19]. Architectures for custom products are derived from PLA. To enable such derivation, PLA must accommodate concepts of variability.

ADLs provide notational frameworks for architecture modeling. Examples of popular ADLs include Acme [4], C2 [5], Darwin [6], Rapide [7], UniCon [8], and Wright [9]. Among few ADLs that support PLA modeling we found xADL 2.0 [1], Koala [10] and ABC/ADL [25]. ADLs for PLA support *variation elements* to model product-specific differences. Typical variation elements are optional component/interface/connector/link and alternative component. During PLA customization, the application architect determines whether an optional element should be selected or not, and which variant of an alternative element is to be included in the product architecture.

At the level of variation elements, the number of possible customizations explodes in combinatorial way. Any two interacting components may involve as many as nine variation elements, namely the two components, one connector, four interfaces and two links; the two components can further be alternative (with variants). Among 2^{11} combinations of those variation elements, only a small number is valid. As an example, Figure 2 shows an invalid design for the optional feature ***Log***. In comparison with the design in Figure 1, we can see that the mandatory connection between optional component ***Log*** and its client ***OnlinePayment*** component are used in Figure 2. Design of Figure 2 does not correctly reflect the interaction between the optional component ***Log*** and its client ***OnlinePayment***. Furthermore, if the component ***Log*** is removed during customization, it is likely to overlook the necessary customization to its clients, for example, leaving a dangling link in the product architecture. For this problem, we expect to identify a set of rules to help eliminate those invalid combinations.

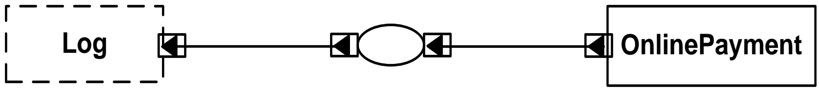

Fig. 2. A meaningless design of optional feature ***Log***

The other problem is that different valid combinations of variation elements may have similar structures but different meanings. For example, design of Figure 3 is similar to Figure 1, but the meaning is quite different. In Figure 1, mandatory ***OnlinePayment*** service is provided with or without logging. On the contrary, in the design of Figure 3 ***OnlinePayment*** is an optional service, but it must be provided with logging if ***OnlinePayment*** is selected.

In the next section, we introduce the concept of variation construct to combat the

above problems.

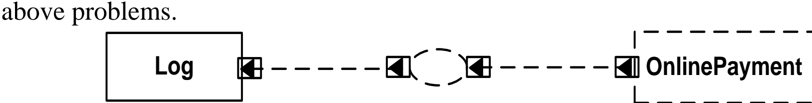

Fig. 3. A design of optional feature ***OnlinePayment***

3 PLA Meta-Model and Variation Combination Rules

This section first presents a PLA meta-model defining architecture modeling concepts found in many ADLs. Then based on the meta-model, we derive a set of rules that can be used to identify valid combinations of variation elements.

3.1 PLA Meta-Model

To present our approach in ADL-independent way, we define essential PLA modeling elements as a meta-model shown in Figure 4. Architecture modeling elements include components, connectors, interfaces and links. Each component can have a possibly empty set of supplier interfaces and client interfaces. A connector must be connected to at least one supplier component and one client component via its interfaces. Each link establishes connection from a client interface to a supplier interface. The additional constraint to links in the meta-model prescribes that links can only be established between component interfaces and client interfaces. That means components can only be composed via connectors.

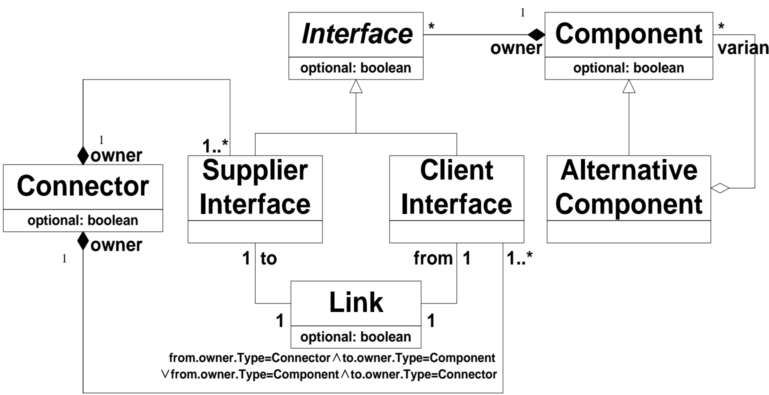

Fig. 4. A meta-model of product line architecture

Variability in PLA is represented by optional and alternative architecture elements. Components, connectors, interfaces and links can be optional. For simplicity, we assume that only a component can be alternative. Some ADLs (e.g., xADL 2.0 [1]) include alternative connectors , but they are not frequently used.

An optional component or connector may be selected or removed during PLA

customization. When de-selected, all corresponding interfaces of an optional element must be removed. It should be noted that a component/connector being optional does not imply that its interfaces must also be optional. An optional interface can be selected or removed for its owner component or connector: a removed supplier interface means the service is not provided; a removed client interface means the service request is not activated. Optional links usually are used together with optional components, connectors or interfaces to represent optional interactions. An alternative component has 0 to multiple variant components for customization. If no predefined variant is provided, it can be regarded as an abstract component with product-specific implementation that can only be provided in application engineering.

3.2 Rules to identify valid combinations of Variation Elements

Some valid combinations of variation elements are implied by PLA meta-model. For example, the rules annotated for link in Figure 4. Here we further explore constraints related to combination of variation elements as shown in Table 1. The predicate op(*ele*) denotes that architectural element *ele* is optional, and alt(*ele*) denotes that *ele* is alternative.

Table 1. Rules for variation constructs

R1	$link \in \text{Link} \wedge (\text{op}(link.from) \vee \text{op}(link.to) \vee \text{op}(link.from.owner) \vee \text{op}(link.to.owner)) \rightarrow \text{op}(link)$
R2	$link \in \text{Link} \wedge \text{op}(link) \rightarrow \text{op}(link.from) \vee \text{op}(link.from.owner)$
R3	$serIF \in \text{SupplierInterface} \wedge \text{op}(serIF) \wedge serIF.owner.\text{Type=Connector} \rightarrow \exists\, serIF' \in \text{SupplierInterface} \wedge serIF' \neq serIF \wedge serIF'.owner = serIF.owner$
R4	$cliIF \in \text{ClientInterface} \wedge \text{op}(cliIF) \wedge cliIF.owner.\text{Type=Connector} \rightarrow \exists\, cliIF' \in \text{ClientInterface} \wedge cliIF' \neq cliIF \wedge cliIF'.owner = cliIF.owner$
R5	$serIF \in \text{SupplierInterface} \wedge \text{op}(serIF) \wedge serIF.owner.\text{Type=Component} \rightarrow \text{alt}(serIF.owner)$

PLA variation constraints are derived from general observations about what forms a correct configuration of variation elements during customization. First, links cannot be dangling, i.e., a link must not be selected unless either of the interfaces it connects to is selected during PLA customization. This constraint makes the first rule (R1) in Table 1, stating that for each link *link*, if either of the interfaces or either of the components it connects to is optional, it must be optional.

Second, if we select a client interface then we must also select a corresponding service provider for it. The second rule (R2) in Table 1 is derived from this point: if a link is optional, the interface or the component at its client side must be optional.

Third, to ensure a connector connects to at least one supplier component and one client component, a supplier (client) interface of a connector must be mandatory if it is the only supplier (client) interface. This constraint is reflected by the third and fourth rules (R3, R4) in Table 1.

Fourth, an optional supplier interface means the service can be provided or not. This usually tells that the owner component has several variants and they are not consistent in providing the service. The fifth rule (R5) describes this constraint.

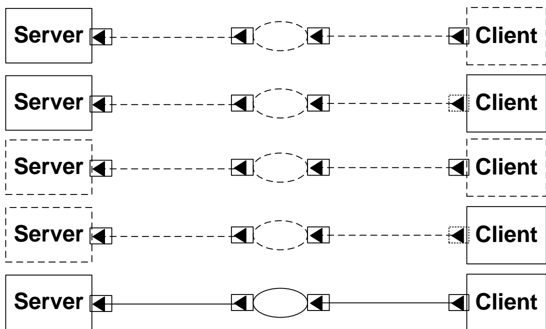

Fig. 5. The set of variation constructs with the constraint of one-to-one connector

With these rules, we can formally eliminate most invalid combinations of variation elements. For example, the combination in Figure 2 can be determined to be invalid according to rule R1. We developed a tool to identify valid combinations of variation elements, i.e. variation constructs. For a composition between two components without alternative components, the tool produces 44 variation constructs, and all of them are meaningful according to our validation.

It should be emphasized that these rules are only the minimum constraint set with the PLA meta-model given in Figure 4. Given additional specific architectural styles, more rules can also be extended. For example, if each connector is restricted to be shared by exact two components, a much stronger rule telling that connector interfaces cannot be optional can be derived from the third and fourth rule. Then we can identify a set of 5 variation constructs as shown in Figure 5 according to rule R1-R5.

4 Variation Constructs in PLA

In our method, a variation construct is defined as a group of architectural elements in a PLA that represent a meaningful functional variation point in the architecture and must be managed together as a whole in PLA evolution and customization. In PLA design, variation constructs can be instantiated and composed with other variation construct instances. And instances of variation constructs can overlap, that is, one architectural element can belong to two or more instances of variation constructs.

In this section, we first introduce the principles in identifying useful variation constructs, then present those identified variation constructs according to the categorization of optional construct and alternative construct with some examples. Those examples are extracted from the Wingsoft Financial Management System Product Line (WFMS-PL). The initial version of WFMS-PL was developed in 2003 and it has evolved into a product line with more than 100 customers today, including major universities in China such as Fudan University, Shanghai Jiaotong University, Zhejiang University [13].

4.1 Principles in Identifying Useful Variation Constructs

The rules given in 3.2 specify the minimum constraint set of valid combinations of variation elements. For example, as mentioned in 3.2, we can identify 44 variation constructs for a composition between two components without alternative components. They are still a little more than a compact variation construct set for reuse, and we also found some of those variation constructs are not as useful as others. Therefore for those combinations satisfying all the rules, we still need to identify useful variation constructs with clarified variability meaning for developers to use.

There are some principles that can be used as guidance to identify variation constructs. These principles involve both the syntax aspect and semantic aspect, i.e. the intention of the design, of architectural variability modeling.

The first principle (P1) is reusing modeling elements among connections as many as possible. In our construct library, both the link and the connector can be reused. The third variation construct of Figure 7 presents such an example. Its link between the connector and *ExceptionHandle* component is used by two connections. Not only can this principle reduce the variation elements but it can also enable the connector to coordinate the interactions between *ExceptionHandle* component and the other parts of the system.

The second principle (P2) is to clearly distinguish the two different intentions of optional interface in alternative components. The first intention is to represent the optional attribute of the component's function that is consistently involved in all the variant components. The second one is to indicate that the interface is only provided by a part of the variants of the alternative component. As the two different intentions have quite different instructions on component implementation, they should be distinguished in the PLA design and reflected in different constructs. For example, for the latter case, the dependence between the variant component and the other parts of the system should be eliminated.

The third principle (P3) is that optional interactions between two components should be modeled at the client side using optional client interface rather than the supplier side. This kind of optional interactions usually reflect internal variation points within components. Then a supplier interface can only be optional when the interface is introduced by different behaviors of the variants of an alternative component (see rule R5 in Table 1). In that case, the optional supplier interface is defined to eliminate the dependency between an alternative component and other parts of the system.

Guided by these principles, we propose to identify different kinds of meaningful combinations of variation elements as architecture-level variation constructs, and further support PLA variability design and customization by variation constructs rather than individual variation elements. In this paper, we report our initial study on the variation constructs between two interacting components. The constructs are divided into two categories according to the two typical kinds of architectural variation points, i.e. optional constructs and alternative constructs. In the following subsections, we will describe variation constructs of the two categories with some typical examples from our case study.

4.2 Optional Constructs

An optional construct means it can be selected or removed during PLA customization. In the PLA, an optional construct is usually modeled using optional component and optional composition between components. An optional composition denotes that the interaction between two components can be selectively included in specific products, and is usually modeled as optional interfaces, connectors and likes.

Figure 6 shows three examples that use optional component to model an optional construct. The first example represents the situation that optional component provide optional feature for the system, implying that only some products require to log the online payment history. According to rule R1 and R2 (see Table 1), an optional component ***Log*** is introduced, together with an optional connector and two optional links, to model the optional feature ***Log***. These two rules are elementary, and almost all the constructs involve the application of them. Note that the interface in ***Log*** component is mandatory. However, the interface in its client component ***OnlinePayment*** is optional, since the availability of ***Log*** is not known until the customization phase. According to the third principle (P3), this kind of variation points is better to be modeled in the client side rather than in the supplier side. Optional components sometimes may invoke other parts of system, which is the case in the second example. The optional component ***AdditionalCharge*** reflects the fact that only some products need to charge additional fee. Since the charge strategy is very complex, ***AdditionalCharge*** delegates to another component ***CommonOperation*** to calculate the tax rate.

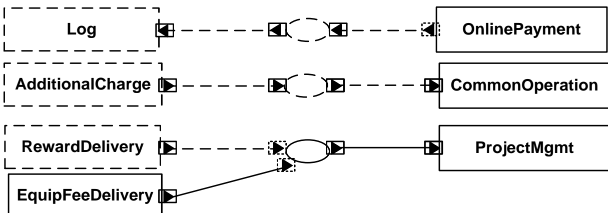

Fig. 6. Optional constructs by optional component

The connectors in the above two examples are both optional, and only serve the connection for the optional component. In some cases the connector for optional construct is at least used once, it should then be mandatory. As shown in the third example of Figure 6, ***RewardDelivery*** provides online delivery of additional reward for staffs and students in university. If selected, the component ***RewardDelivery*** depends on the service provided by the component ***ProjectMgmt*** to deduct the amount from corresponding project fund. As some universities tend to deliver salary offline, ***RewardDelivery*** is designed to be optional. The amount deduction service provided by ***Project*** is also required by other mandatory components such as ***EquipFeeDelivery***, so the connector is at least necessary by one component. According to the first principle (P1), the connector is modeled to be mandatory. The only entrance of amount deduction service also brings benefit of easy management for payment control.

Optional composition represents finer-grained variability than optional component.

Figure 7 shows three examples that use optional composition to model optional construct. In the first example, ***Budget*** and ***ProjectMgmt*** are two mandatory business components for budget management and project management respectively. ***Budget***, when is executed for project budget making, involves an internal variation point of whether performing balance control (provided by ***ProjectMgmt***) or not. According to P3, the optional client interface of ***Budget*** and related optional connector and links are employed to model this internal variation point. The component ***Initiation*** in the second example is responsible for initializing the system's basic information, such as the mode of payment and the student state. The initialization of the student state is optional, which is also a fine-grained variation point and modeled as an optional interface in component ***Initiation***. The difference from the first example is that the service provided via the supplier interface of ***Utility*** is also required by other mandatory components. According to R3 that a supplier (client) interface of a connector must be mandatory if it is the only supplier (client) interface, the interface in the connector which links to ***Utility*** must be mandatory. So according to P1, the connector and the composition to ***Utility*** are modeled as mandatory elements. Optional composition can also exist between optional components. As an interface will be removed together with its owner, the interfaces of an optional component usually do not need to be modeled to be optional. But sometimes an interface of an optional component should still be modeled to be optional to represent another finer-grained variation point. As shown in the third example in Figure 7, WFMS-PL can support normal payment as well as web service payment. The exceptions thrown by the component ***WebServicePayment*** can be handled inside itself or by the ***ExceptionHandle*** component. That means the client interface of ***WebServicePayment*** may be removed even when ***WebServicePayment*** is selected, so the interface and related link and connector interface are modeled to be optional.

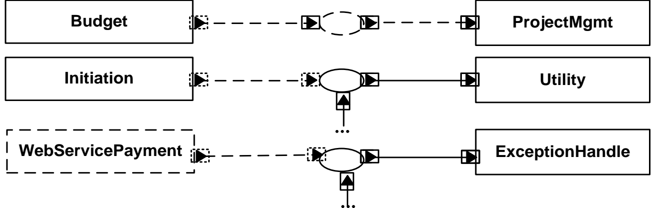

Fig. 7. Optional constructs by optional compositions

4.3 Alternative Constructs

An alternative construct means that it can be replaced by different variants during application customization. Figure 8 presents three examples that use alternative components to model an alternative construct. Alternative component is shown in a large box containing variants in smaller boxes. The first example describes the case that all the variants of a component exhibit the same interface. WFMS-PL support many banks that the application engineers can configure and adopt a subset of them in their product. The interfaces of different banks are the same as each other. There is no need to introduce additional variation elements to the ***Bank*** component for this

alternative construct since the selection of different variants does not impact the rest of the system.

However, as shown in the second and third examples in Figure 8, there are also some cases in which only part of the variants of an alternative component provide certain services. For example, as shown in the second example of Figure 8, in WFMS-PL there are three different modes of fee payment, i.e. ***ByItem***, ***ByYear*** and ***ByYearOrder***. ***ByItem*** means user can pay their fee item by item. ***ByYear*** means user can only pay the fee year by year (each fee item belongs to one year) in any order. ***ByYearOrder*** means user can only pay year by year following the time order. Only the variant ***ByYearOrder*** may raise exception and requires the service provided by ***ExceptionHandle***. According to P2, the alternative component must use optional interfaces to identify those interfaces that are not exposed by all of its variants [11]. Only if the variant ***ByYearOrder*** is selected, the optional interface in ***FeeItemSelection*** will be selected as well. Similarly, in the third example in Figure 8 only the variant ***Operation*** of the alternative component ***LockFeeItem*** provides the interface for ***OnlinePayment***. In contrast to general principle P3 that the variability is better to be modeled in the client side rather than in the supplier side, it is interesting to note that the variability introduced by the ***Operation*** variant of the ***LockFeeItem*** component is modeled on the supplier side. This is because that this optional interface is introduced for eliminating the unnecessary dependencies between the other variants of the component ***LockFeeItem*** (e.g. ***Delegation***) and the rest of the system (e.g. ***OnlinePayment***). This kind of variability cannot be moved to the client side.

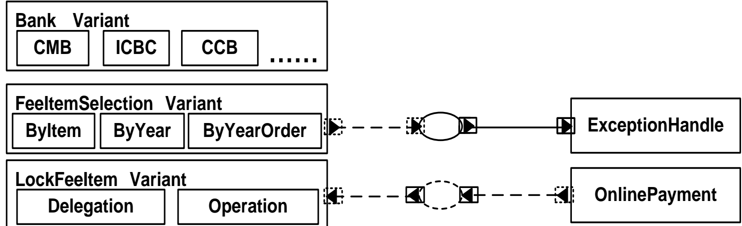

Fig. 8. Alternative Constructs

5 Implementation and Case Study

5.1 Implementation

In order to validate the proposed approach, we developed an Architecture Centric Software Product Line (ACSPL) tool that supports construct-based variability management, as shown in Figure 9. The tool is equipped with a graphical editor for PLA modeling (the upper right editor in Figure 9). In the lower left view, the tool lists all the instances of variation constructs in the PLA under development. In the construct view, the construct instances are divided into two categories, i.e. optional construct and alternative construct. Each construct instance contains all its related variation elements. The user can apply adaptive, corrective, and perfective [12] operations to the constructs and their contained variation elements, such as adding or

removing a construct, changing the construct type.

(1) Add a variation construct.

This situation happens when developer is designing a new variant feature. In ACSPL, developer has two options modeling this requirement. The first one is the traditional way in which developer edits the variation elements individually. The second one is to select an intention such as optional construct by optional connection and choose from candidate constructs provided by the tool. Then a skeleton variation construct will be added by the tool.

(2) Remove a variation construct

Removing a variation construct does not mean to remove the construct's elements but the elements' variability. With ACSPL, developer just needs to operate on one construct to eliminate the variability of all the relevant variation elements. After this operation, optional element becomes mandatory and alternative component to selected variants.

(3) Change the type of variation construct

This situation happens in PLA evolution [24]. For example, the need for more fine-grained control over a variation point could requires changing the type of variation construct from optional component construct category to optional composition construct category. In traditional way, in order to keep consistency, developer firstly needs to remove all the related elements' variability and then applies the change operations. ACSPL have documented the structures of the constructs, developer can be ensured the operation is consistency applied and also can clearly see the change impact before bringing it into effect.

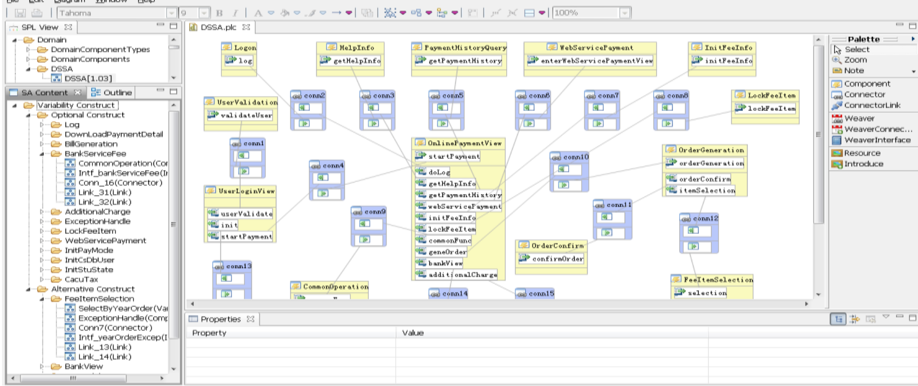

Fig. 9. Architecture Centric Software Product Line (ACSPL) Tool

5.2 Case Study

We have developed and maintained Wingsoft Financial Management System Product Line (WFMS-PL) using the proposed construct-based variability management approach. We have identified 21 key variation constructs during the development and evolution of WFMS-PL. The 22 variants of the key constructs have also been identified. Among the current 43 constructs in the catalog, the instances of 7 constructs, which is shown in Table 2, account for 70 percent of construct instances

identified in the WFMS-PL.

The variation constructs raise the level of abstraction in PLA design and customization. We no longer model and manage variability at the detailed level of variation elements and their relationships. We only need to manage 53 variation constructs and 42 relationships among them, in comparison with the 217 individual variation elements and 279 relationships among them.

Table 2. Different Constructs used in WFMS

Variability Constructs	#Instance
	11
	7
Variant Component variant1 variant2	6
	5
Variant Component variant1 variant2	4
	3
	3
Others	14

In the past, in order to keep the consistency of the product architecture, variation elements usually inherit variability from its contained or linked elements. For example, a link connect to an optional interface is inherited optional. Thus they are selected or unselected together in the customization phase. This can certainly solve a part of inconsistency evolution problem. But not all the relations between variation elements are so transparent. As shown in Figure 10, the component *Initiation* is responsible for initializing the system's basic information, such as the mode of payment and the student state. Both the initialization of the student state and the mode of payment are optional, which is modeled as two optional interfaces in component *Initiation*. Since the connector is mandatory, it takes time to figure out the correspondent relations between two sides of the connector, which makes the evolution error prone. If the construct is introduced and becomes the management unit, in the lower left view of Figure 9, all the correlated variation elements are listed explicitly and this alleviates the inconsistency problem. Compared with traditional way, variation construct also gains benefits that variation elements in a construct can be different granularity. If the interface's variability is inherited from its contained components, they will be selected or unselected according to their components. But in some cases, the optional component is included and its optional interface is unselected. On the other hand, if the construct is the basic management unit, a variation element can be contained by many constructs and unless all the constructs are unselected the

variation element is unselected. Thus the elements in one construct may have different bound conditions and granularities.

The core assets of the WFMS-PL were designed and implemented by few *domain engineers*. The maintainer of the system is changed continually. Thus it takes significant amount of time for a new member of the project team to familiarize himself with the system and all possible variants. Through our case study, we find out that there exists certain connection between the construct and the implementation. For example, construct like Figure 1 is implemented through reflection mechanism in WFMS-PL. Thus with our construct, developer can not only understand the architecture easily but also get some insights of the implementation, which can greatly speed up their learning process.

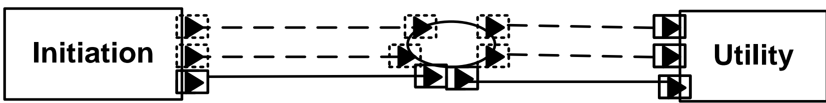

Fig. 10. An error prone situation of customization

Furthermore, the variation constructs help to reveal the distinctive meanings of the variability designs in PLA and eliminate the ambiguity. Take the two construct instances in Figure 11 as an example. The two instances are syntactically same in that they involve the same set of variation elements. However, the optional interface in alternative component *Bank* is exposed by all its variants, such as *CMB* and *ICBC*. This optional interface is introduced because some customers have to pay the fees for the bank online service while others do not. In contrast, the optional interface in alternative component *FeeItemSelection* is introduced because it is only exposed by the variant *SelectByYearOrder*. The variant *ByYear* and *ByItem* do not provide this interface. Clearly, the semantics of the two cases are completely different. In fact, they are the instances of the two different variation constructs from optional construct category and alternative construct category respectively.

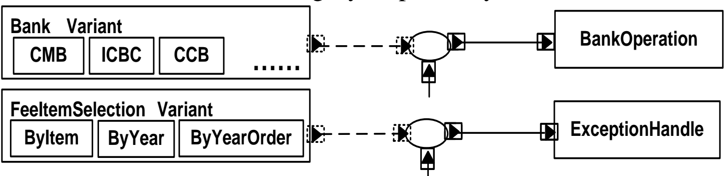

Fig. 11. Syntactically same but semantically different

Although we need more systematic quantitative study, our experience indicates that, by categorizing the variation constructs and providing the tool support, the proposed approach provides not only an efficient way to document and manage the variation points in PLA, but also help to flatten the learning curve of the new comers who joined the project later and reduce the errors due to the inconsistent modifications to variation points.

6 Related Work

As a fundamental aspect of software product line, variability management has attracted a lot of research. Among them, some focus on the development of variability model, e.g. OVM (Orthogonal Variability Model) [14] and PuLSE [15]. The others aim at variation implementation mechanism, e.g. XVCL [16]. However, architecture-level variation management is much less explored.

In the past two decades, many architecture description languages (ADLs), e.g. xADL2.0 [1] and ACME [4], have been proposed to support formal architecture modeling and analysis. In these ADLs, the architectural elements, such as component, connector, interface (port) and link are commonly used to model software architecture. To support PLA, some ADLs, e.g. xADL2.0 [1], further introduce the variability mechanisms to these architectural elements. However, such ADLs lack of mechanism to model structural variation constructs. Furthermore, they do not provide any effective mechanism to manage the large amount of variation elements and the possibly even larger number of dependencies between these variation elements [17].

In order to improve architecture-level variability management, Hendrickson et al. [18] have proposed the change-based methods for PLA modeling, in which PLA variations are modeled by change sets and relationships. However, they acknowledged that state-based PLA modeling, in which variability is represented by variation elements inside the architectural specification, is still the mainstream of product line architecture design [19]. In this paper, we propose a construct-based method, which provides state-based variation mechanism for PLA modeling.

For effective management and composition of architectural variability, Neil Loughran et al [20] have proposed a variability modeling language (VML). VML supports compositions involving both fine-grained and coarse-grained variabilities in an orthogonal fashion. With our method, VML will reference variation constructs instead of primitive variation elements. Thus our approach can complement this approach rather than replace it.

There has been some work on using patterns to model variability in product lines. The proposed patterns [21, 22] work close to the system implementation. For example, they use adapter pattern to model the alternative function [21]. In contrast, our variation constructs are at a higher level of abstraction. We focus on the variation constructs in component based product line architecture, consisting of components and their interactions represented by interfaces and connectors.

7 Conclusion and Future Work

In this paper, we presented an architectural variability modeling method by structural construct for the design and customization of software product line architecture (PLA). With our method, the architect can conceive and model the variability design for each variant feature by structural construct rather than individual variation element. The proposed approached has been implemented in a software product line tool and has been evaluated on an industrial financial system. Our preliminary results indicate that the raising of variability modeling level by structural constructs improves PLA design and customization by reducing the complexity and inconsistency of the variability

modeling.

We plan to extend this work in two directions. First, the current construct catalog has been extracted and validated on the WFMS-PL. But we believe that they would be applicable in other software product lines, since the principles underlying these constructs are not specific to a given system. We plan to further refine and enrich the current construct catalog with more subject systems. Furthermore, the current variation constructs involve only two components and their interactions. We are also interested in extending the concept to the constructs involving multiple components.

Second, we plan to explore the backward and forward traceability between the architecture-level variation constructs and the variation points in the analysis models, such as feature models [23], and in the product line implementations, such as XVCL [16]. We believe that architecture-level variation constructs can serve as an intermediate layer that helps to trace and manage the variations across different levels of abstraction. We would like to investigate if they can facilitate the consistent feature-driven derivation of application products. Furthermore, we want to investigate if they can improve the evolution of software product line, e.g. helping to populate variation points with new variants, to prune old, no longer used, variants, as well as to distribute new and/or changed variants to the already installed products.

Acknowledgments. This work is supported by National Natural Science Foundation of China under Grant No. 90818009, Shanghai Committee of Science and Technology, China under Grant No. 08DZ2271800 and 09DZ2272800, Shanghai Leading Academic Discipline Project under Grant No. B114.

References

1. Eric M. Dashofy., André van der Hoek., Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology, 2005. 14(2):p. 199-245.
2. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture De-scription Languages. IEEE Transactions on Software Engineering, 2000. 26(1): p. 70-93.
3. P. Clements and L.M. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, New York, New York, 2002.
4. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research (CASCON) 1997.
5. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering (FSE), 1996.
6. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. Proceedings of the 4th Symposium on the Foundations of Software Engineering, 1996.
7. D.C. Luckham and J. Vera. An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, 1995. 21(9): p. 717-734.
8. M. Shaw, et al.. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 1995. 21(4): p. 314-335:
9. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. ACM Transactions on

Software Engineering and Methodology, 1997. 6(3): p. 213-249.
10. R. van Ommering, et al.. The Koala Component Model for Consumer Electronics Software. Computer, 2000. 33(3): p. 78-85.
11. Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.. Mae—a system model and environment for managing architectural evolution. ACM Transactions on Software Engineering and Methodology, 2004. 13(2):p. 240–276.
12. E. B. Swanson. The dimensions of maintenance. Proceedings of the 2nd International Conference on Software Engineering (ICSE), pages 492–497, 1976.
13. Ye, P., Peng, X., Xue, Y. and Jarzabek, S.. A Case Study of Variation Mechanism in an Industrial Product Line. Proceedings of the 11th International Conference on Software Reuse (ICSR), 2009.
14. Pohl K., Metzger A.. Variability management in software product line engineering. Proceedings of the 28th International Conference on Software Engineering (ICSE), 2006.
15. K. Schmid and I. John. A customizable approach to full lifecycle variability management, Science of Computer Programming, 2004. 53 (3), pp. 259–284.
16. Jarzabek, S., Bassett, P., Zhang, H. and Zhang, W.. XVCL: XML-based variant configuration language. Proceedings of the 25th International Conference on Software Engineering (ICSE), 2003.
17. Deelstra, S., Sinnema, M. and Bosch, J. Experiences in Software Product Families: Problems and Issues during Product Derivation. Proceedings of the 3st Software Product Lines Conference (SPLC), Boston, Aug. 2004, LNCS 3154, Springer-Verlag, pp. 165-182.
18. Hendrickson, S.A., van der Hoek, A.. Modeling Product Line Architectures Through Change Sets and Relationships. Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, USA, pp. 189–198, 2007.
19. Nicolás López, Rubby Casallas, André van der Hoek. Issues in Mapping Change-Based Product Line Architectures to Configuration Management Systems. Proceedings of the 13th Software Product Lines Conference (SPLC), pp. 21-30, 2009.
20. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.. Language Support for Managing Variability in Architectural Models. Proceedings of the 7th International Symposium on Software Composition (SC). Volume 4954 of LNCS., Budapest (Hungary) (March 2008) 36–51.
21. B. Keepance and M. Mannion. Using patterns to model variability in product families. IEEE Software, 1999. 16 (4), pp. 102–108, July/August 1999.
22. Jiang, J., Ruokonen, A., Systä, T.. Pattern-based variability management in Web service development. Proceedings of the 3rd European Conference on Web Services (ECOWS 2005), 2005.
23. Kang, Kyo C., Sholom G. Cohen, James A Hess, William E.Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
24. Peng, X., Shen, L., Zhao, W.. An Architecture-based Evolution Management Method for Software Product Line. Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE), 2009.
25. Mei, H., Chen, F., Wang Q., Feng Y.. ABC/ADL: An ADL Supporting Component Composition. Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM), 2002.