An Architecture-based Evolution Management Method for Software Product Line

Xin Peng, Liwei Shen, Wenyun Zhao
School of Computer Science, Fudan University, Shanghai 200433, China
{pengxin, 061021062, wyzhao}@fudan.edu.cn

Abstract

In software product line (SPL) development, evolutions occur in core assets and application products. How to ensure their alignment in evolution is a big challenge. Products in an SPL share a reference architecture, which centers in SPL development and evolution, so architectural evolution management is a natural and essential choice for SPL. In this paper, we propose an architecture-based evolution management method for SPL, in which both architecture and component evolutions are supported. An integrated version model for both core assets and application products is proposed. Based on the model, the method provides evolution processes for architectures and components, both supporting forward customizations and backward feedbacks by merging and synchronization. The prototype tool for the method has been developed on the open-source version control system Subversion, and preliminary application has shown that it can effectively support SPL evolutions.

1. Introduction

In SPL, there are both domain-level (domain engineering) and product-level (application engineering) developments with different goals and disciplines. These two kinds of development activities are often inclined to evolve to different directions if there is no effective coordination. For example, application engineers may decide to make architectural adaptations incompatible with the reference architecture or directly modify domain components under demands of product customers. If this kind of deviations accumulates, the organization will lose the control on the product line gradually. Therefore, successful SPL engineering requires management and coordination of two kinds of development activities to meet the organization's overall business goals [1].

Software configuration management (SCM) is the discipline of managing the evolution of complex software systems [2]. The discipline enables us to keep control and track software changes, and is an integral part of any software development and maintenance activity [3]. In traditional software development, SCM is performed within each project. However, in SPL, evolution management of application-engineering projects and the domain-engineering project should be coordinated and unified to maintain the integrity and consistency of the SPL. The overall coordination must ensure that the products and core assets remain aligned with each other [1].

Products in an SPL share a reference architecture, which specifies the common structure of the products and centers in the development and evolution of both core assets and application products. Architectural SCM acknowledges the central role that the software architecture plays in software development and maintenance [2]. Therefore, architectural SCM is an effective means for evolution management in SPL. There have been some related works on evolution management for SPL architecture, e.g. the xADL-based works ([4][5][6]). These works focus on evolutions of architectures only and do not provide supports for component evolutions, evolution integration and product release, etc.

In this paper, we propose an architecture-based evolution management method for SPL. The method provides coordinated and unified evolution management for SPL, which can keep the continuous optimization of core assets, and at the same time support the implementation of customer requirements in each product. On the other hand, the method supports evolutions of both architecture/component specifications and component implementations. In the method, evolution managements for architectures and components are separated, and a comprehensive version model for both core assets and products is proposed to form the basis of evolution integration and release configuration.

The remainder of this paper is organized as follows. Section 2 introduces some related work and compares our works with them. Section 3 presents the evolution management method. Section 4 presents a case study and evaluates our method. Finally, we draw conclusions and discuss future work in section 5.

2. Related work

Traditional SCM tools are designed with the intention of versioning a single product, so they do not have facilities to support forward and backward change propagations [7]. Van Gurp et al. [8] propose to

combine product derivation and variability management based on existing version management tools (e.g. Subversion [9]). Yu et al. [3] propose an evolution-based SCM model for SPL, but no detailed introduction on architecture and component evolutions are reported. Thao et al. [7] present a SCM system MoSPL for product derivation in SPL. MoSPL provides version management at the component level, and explicitly manages logical constraints and derivation relations among components, thus enabling the automatic propagation of changes in core assets to products and vice versa. Their method concentrates on component-level derivation and evolution only.

The xADL group has a series of works on evolution management for SPL architecture. They present Ménage, the xADL 2.0 based environment for managing evolving SPL architectures in [6]. The tool provides supports for architectural element versioning and reference architecture customization. Their architecture differencing and merging method is presented in [5]. However, evolution synchronization and component evolutions are not mentioned.

Our method adopts xADL 2.0 to represent both reference and application architectures also. However, our evolution management method differs from theirs at several aspects: evolutions of both abstract architecture models and component implementation are supported; both evolutions of core assets and application products are involved with periodic synchronizations. Furthermore, in architecture merging, our method adopts the policy of variability abstract on the differences among the reference architecture and application architectures, not the all-included merging in [5].

3. Our SPL evolution management method

3.1 xADL 2.0

xADL 2.0 [4] is a highly-extensible, XML-based architecture description language, which includes a set of schemas to describe the architecture of a single software system or a product line. The most important part is the **Structure&Type** schema, which is used to describe basic architectural elements at design time, including components, connectors and links. Each component in architecture can have a component type describing the type information of the architectural components, including signatures, etc.

Architectural variability for SPL is supported by the **Options** and **Variants** schema. Options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of a group of elements [4]. Variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements [4]. Each optional or variant element is accompanied by a guard condition to determine the inclusion or exclusion of it. Readers can refer to [4] for detailed introductions to xADL 2.0.

Figure 1 depicts an xADL-style reference architecture of the online book shopping product line. In the architecture, there are optional component ***AcctMgtUI*** and ***AcctMgt*** and variable component ***Payment*** and ***Discount***. ***Payment*** has two variants of ***PayByVirtualCur*** and ***PayByCreditCard*** for two different modes of payment. ***Discount*** has no variants, implying that each application can have different discount policy, so it is an abstract component to be instantiated in application engineering. Guard conditions for these optional and variant components are also listed in Figure 1. It can be seen that payment mode (represented by the symbol ***payMode***) is the main variation point. And variation constraints are implied by guard conditions: if setting ***payMode*** to be ***virtualCur*** then components for account management (***AcctMgtUI*** and ***AcctMgt***) should also be bound.

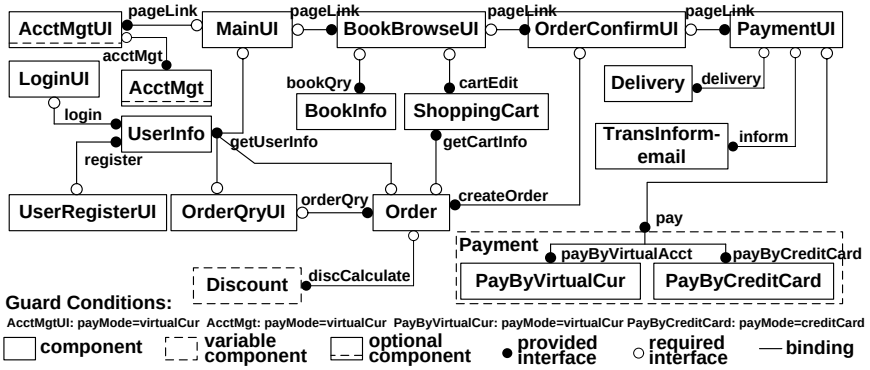

Figure 1. Reference architecture of the online book shopping product line

3.2 SPL version model

The version model of our method extends the structure model of xADL 2.0 schemas of **Structure&Type**, **Options** and **Variants**. The model is depicted in Figure 2, in which grey boxes represent elements from xADL 2.0 and others are our extensions.

From the model, we can see that both architectures and components are versioned entities. In each product line, there is only one reference architecture (**RefArch**), and all the application architectures (**AppArch**) are derived from it. Both **RefArch** and **AppArch** can have multiple versions, and each version is composed of a set of **Component**, **Connector** and **Link**. Besides, there are architectural variations (optional and variant components) and variation constraints in each **RefArch** version. Each **AppArch** version may be synchronized with a **RefArch** version or not, representing the independent evolutions and periodic synchronizations of **RefArch** and **AppArch**. By synchronization, we mean that the adaptations of an application asset are within the variability scope of corresponding domain asset. Similarity, there are both domain and application components, their versions and derivation/synchronization relationships between them.

In our method, component specifications are separated from component implementations as independent versioning entities. Each component implementation declares a component specification as its type and then each version of it will implement a specification version, representing that the component implementation complies with the specification. To distinguish the evolutions of architectures and components, we assume that each **ComponentType** in xADL 2.0 refers to a component specification version in our model, thus component implementations are completely separated from architectures.

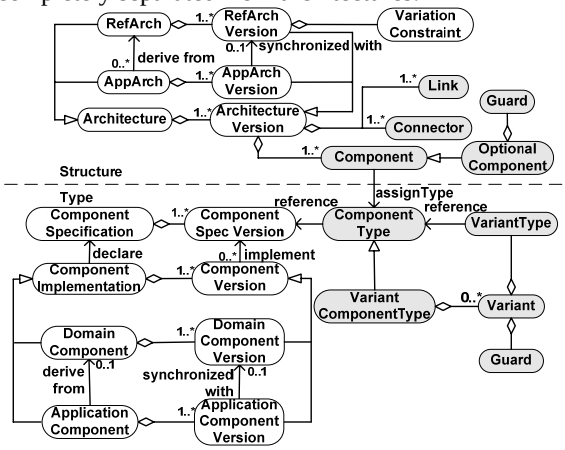

Figure 2. SPL version model

3.3 Evolution process

In our method, core assets and products can evolve independently. Temporary deviations are allowed, and periodic synchronizations on both architecture and component level will be performed to reunify core assets and application products.

Architecture evolution process in our method is presented in Figure 3. We can see that besides independent reference or application architecture evolutions, there are also cross evolution paths, including architecture derivation, architecture merging and synchronization. The first version of an application architecture is always derived from the latest version of the reference architecture, e.g. **Aa1.0** is derived from **Ra1.1** and naturally they are synchronized. After that, the application architecture can evolve independently (e.g. **Aa1.1** and **Aa1.2**). On the other hand, reference architecture may also evolve for design optimization or new features (e.g. **Ra1.2**). After some time, periodic merging is performed among current versions of the reference architecture and all application architectures to make a new reference architecture version (e.g. **Ra1.3**). This merging propagates architecture evolutions in application products to the reference architecture. After that, synchronization is performed to propagate evolutions of reference architecture back to application architectures. Then, reference architecture and application architectures are synchronized again (e.g. **Ra1.3** and **Aa1.3**).

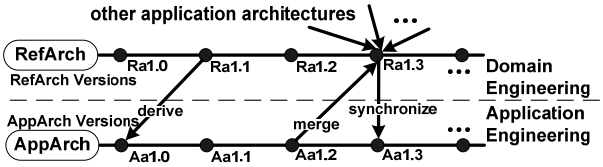

Figure 3. Architecture evolution process

In our architecture-centric evolution management, component evolutions are managed in term of their variability types, as shown in Figure 4. Derived application components are first derived from domain components along with the application architecture derivation or synchronization. It can be derived from a mandatory component, or an optional or variant component that is selected in architecture customization. After derivation, the application component can evolve independently and periodically be merged with corresponding domain component. Application-specific components are first created entirely for an application along with application architecture evolution. It may be a new part added to the architecture or a new variant for variable domain component. It will be evaluated in architecture merging by the domain architect and may be adopted as a domain component if the architectural extension is accepted into the reference architecture. Besides, an application-specific component can also be the application-specific implementation (instantiation) for an abstract domain component. In this case, only synchronization on component specification should be considered in following evolutions, since abstract domain component specifies type information only.

Among all the cross component evolutions, component derivation, adoption and replacement can only occur along with architecture evolutions, while the others (component merging and synchronization) can occur independently.

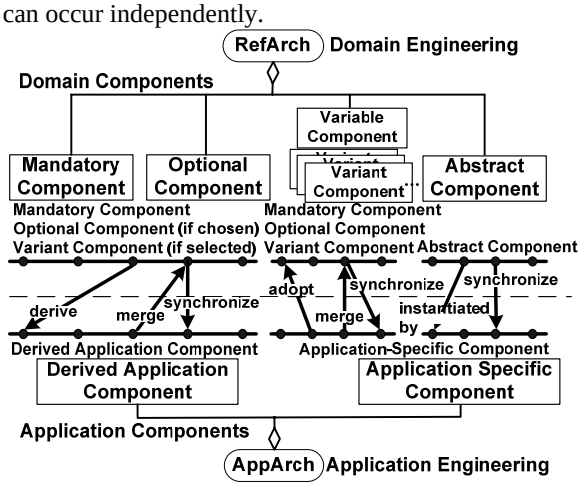

Figure 4. Component evolution process

3.4 Architecture evolutions

3.4.1 Architecture derivation Guard conditions in xADL 2.0 provide a built-in mechanism for automated application architecture derivation, e.g. the xADL environment [6] provides the SELECTOR component. A guard condition is a Boolean expression composed of symbol, value and the comparison between the two parts, i.e. equal to, greater than, etc. A symbol can be used in several guard conditions for different optional or variant components. In architecture derivation, the application engineer will be requested to assign values to all the symbols, then all the optional or variant components can be determined to be bound or not according to the value of their guard conditions.

3.4.2 Architecture merging Architecture merging in our method is performed on architectural differences between reference architecture and application architecture, which can be captured by our SPL evolution management environment. The differences can occur at the structure or type level. For example, removing a component is a structural difference, and replacing a variable component with a new variant is a type difference.

We identify 10 kinds of basic architectural differences, including architectural customizations as listed in Table 1, in which type differences are represented by grey lines. ***VariantBound***, ***OptionalComRemoval***, ***AbsComInstance***, ***NewComLinkToAbs*** are architectural customizations within prescribed scope, so no merging operations are needed. Other cases are differences beyond the variability scope and merging operations will be performed. For example, in the cases of both ***NewComponent*** and ***NewComLinkToAbs***, a new component is added in the application architecture. In ***NewComLinkToAbs***, the component is linked to an application component corresponding to an abstract component in the reference architecture, so it is considered to be part of the instantiation for the abstract component. In ***NewComponent***, the component is linked to non-abstract domain components, so it is considered to be an additional architecture adaptation, e.g. the application engineer may decide to add an logging component to the ***Order*** component shown in Figure 1 for better security.

Merging policies for those architectural differences are listed in Table 1. In our method, differences on component specification mean completely different components (e.g. new variant), while differences on component specification version mean revised component specifications (e.g. adding an interface or interface revisions). It can be seen that in some cases user intervention is needed, e.g. to determine whether merged as mandatory or optional component in ***NewComponent***. After merging, new architectural variation points or functional extensions may be added to the reference architecture, e.g. adjusting mandatory components to be optional, or accepting new domain components from application architectures.

3.4.3 Architecture synchronization After architecture merging, the reference architecture embodies all the application differences by new variation points, which makes it possible to synchronize application architectures. Architecture synchronization is to propagate evolutions in reference architecture, from both itself and other applications, to each application architecture. It can be seen as the re-derivation of application architecture from the new reference architecture version.

The symbols in xADL 2.0 represent business or design options independent of specific variation points, so the customization decisions (symbol value assignments) can be reused. For those newly added symbols, the application engineer will be requested to assign values for them.

Table 1. Merging policies for different kinds of architectural differences

Difference Type	Description	Merging Policy
NewComponent	A **Component** in **AppArch** does not exist in **RefArch**, and the new **Component** links to at least one **Component** that corresponds to non-abstract **Component** in **RefArch**	merged as new mandatory **Component**
		merged as new optional domain **Component**
NewComLinkToAbs	A **Component** in **AppArch** does not exist in **RefArch**, and the new	N/A

	Component links to a Component that corresponds to an abstract **Component** in **RefArch**	
OptionalComRemoval	Optional **Component** is removed in **AppArch**	N/A
MandComRemoved	Mandatory **Component** is removed in **AppArch**	change the mandatory **Component** to be optional
DifComSpecVersion	Non-abstract **Component** has the same component specification in **AppArch**, but with a new component specification version	merged with the domain component specification to make a new version
NewComponentType	Non-abstract **Component** has a new component specification in **AppArch**	merged as alternative **Component** and replace the original mandatory **Component**
VariantBound	Variable domain **Component** is customized to one of its prescribed variant in **AppArch**	N/A
DifVariantComSpecVersion	Variable domain **Component** is customized to one of its prescribed variant in **AppArch**, and the variant has a revised specification (new component specification version)	merged with the variant component specification to make a new version
NewVariantComType	Variable domain **Component** is customized to a new added variant (with new component specification) in **AppArch**	Add the new application variant to the variable domain **Component**
AbsComInstance	Abstract **Component** is replaced by an application-specific component in **AppArch**	N/A

3.5 Component and product evolution

Component-level evolutions include individual component evolution and cross evolution also. Individual component evolution may be due to revision of specification or implementation only. For a component, evolution may be due to new specification it implements (e.g. adding a new interface) or purely an implementation revision (e.g. bug fixing). The former is supported by versioning of component specifications and the management of the implementation relations between component implementations and specifications (see Figure 2). The latter is implemented by file-level evolution management and can be supported by traditional version management system, e.g. Subversion [9] integrated in our implementation.

Among those cross component evolutions depicted in Figure 4, component merging and synchronization are the main problems. In component merging, derived application component versions will be merged into corresponding domain component. It is file-based merging of component implementations, so the merging can be supported by the version merging mechanism in traditional SCM systems. Component synchronization is to propagate new version of a domain component to those application components derived from it. After synchronization, a copy of the domain component will become the current version of the application component.

Product evolution is supported by the product release mechanism. As mentioned before, evolutions on the architectural level and component level are separated in our method. Product release should first choose an application architecture version and then determine versions for all the components involved.

4. Case study and evaluation

Our method has been implemented in the prototype evolution management tool ASCMPL (Architecture-based Software Configuration Management tool for Product Line). It is an eclipse plug-in developed on Subversion [9] and xADL [4] library. ASCMPL provides direct support for architecture and component specification development. File-based evolutions of component implementations are managed by Subversion, and file-level versioning information (e.g. URLs and revisions in Subversion) is referred in component configuration information for integration.

In order to evaluate our method, we conduct a case study on an enterprise product line, i.e. the online book shopping system, with ASCMPL. It is a Java-based web system. Figure 1 shows the initial reference architecture of the product line, in which payment mode is considered as the main variation point (see the symbol ***payMode*** in the guard conditions). Based on this reference architecture, an application engineer derives a product variant and adapts the application architecture to meet application-specific requirements. The adapted application architecture is shown in Figure 5, in which grey blocks represent new application components and dotted blocks represent components with modified specifications.

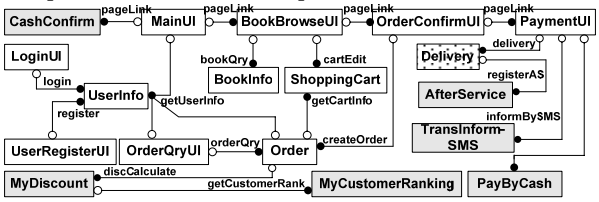

Figure 5. Adapted application architecture

According to our architecture merging method, we identify 9 architectural differences of 7 different types and corresponding merging operations as shown in Table 2. After merging, new variation points are introduced, including optional component ***CashConfirm***, ***AfterService*** and alternative component ***TransInform***. Besides, two new symbols of ***afterService*** and ***inform*** are added and a new candidate value *cash* is added for the existing symbol ***payMode***. Due to the limitation of space, the reference architecture after merging is not presented. In this case, some component-level evolutions are also involved,

e.g. component-level merging between domain component ***Delivery*** and the modified application component ***Delivery***. In following evolution synchronizations, these new features in the reference architecture will be propagated to other applications and their existing customization decisions can be reused. For example, an application with the component ***PayByCreditCard*** can reuse the decision "***payMode*=*creditCard***" in the synchronization, and decisions for new symbols (e.g. ***afterService***) should be complemented of course.

From the case study, it can be seen that for a real software product line, long-term and coordinated evolution management is necessary. Architectural SCM is essential for SPL evolutions, since both of them acknowledge the central role of architecture. Moreover, in order to provide comprehensive evolution management, both specification- and implementation-level evolutions should be supported. Our evolution management method provides an integrated version model for both architectures and components. Based on the version model, the method supports both forward architecture derivation and backward evolution feedbacks by architecture merging and synchronization. It also provides the mechanism to integrate traditional file-based SCM tools to make a comprehensive evolution management for SPL.

Table 2. Architecture differences and merging operations in the case study

Difference	Difference Type	Merging operations
PayByCash	*NewVariantComType*	as a new variant of *Payment* with guard condition "*payMode=cash*"
CashConfirm	*NewComponent*	as a new optional component with guard condition "*payMode=cash*"
AcctMgt	*OptionalComRemoval*	N/A
AcctMgtUI	*OptionalComRemoval*	N/A
MyDiscount	*AbsComInstance*	N/A
MyCustomerRanking	*NewComLinkToAbs*	N/A
Delivery	*DifComSpecVersion*	merged with the domain component to produce a new specification version
AfterService	*NewComponent*	as a new optional component with guard condition "*afterService=true*"
TransInform-SMS	*NewComponentType*	merged with *TransInform-email* to make a new alternative component *TransInform* with guard condition "*inform=SMS*" and "*inform=email*"

5. Conclusion and future work

In this paper, we present an architecture-based evolution management method for SPL. A version model involving both domain and application architectures and components is proposed and evolution processes for architectures and components are presented. The method supports architecture merging and corresponding evolution synchronization. For component-level evolutions, our method supports both specification and implementation evolution.

In our future work, we will try to integrate feature model [10] and other SPL artifacts in the evolution management on certain feature-based traceability mechanism. On the other hand, we will try to integrate the evolution management with our product derivation tool [11] to provide a complete platform for incremental SPL development.

Acknowledgments. This work is supported by National Natural Science Foundation of China under Grant No. 60703092, and National High Technology Development 863 Program of China under Grant No. 2007AA01Z125.

References

[1] P. C. Clements, L. G. Jones, L. M. Northrop, J. D. McGregor. Project Management in a Software Product Line Organization. IEEE Software, 2005, 22(5).

[2] B. Westfechtel, R. Conradi. Software Architecture and Software Configuration Management. In SCM 10, 2001.

[3] L. Yu and S. Ramaswamy. A Configuration Management Model for Software Product Line. INFOCOMP Journal of Computer Science, 2006, 5 (4).

[4] E. M. Dashofy, A. Hoek, R. N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. TOSEM, 2005, 14 (2).

[5] P. Chen, M. Critchlow, A. Garg, et al.. Differencing and Merging within an Evolving Product Line Architecture. In PFE'03, 2003.

[6] A. Garg, M. Critchlow, P. Chen, et al.. An Environment for Managing Evolving Product Line Architectures. In ICSM'03, 2003.

[7] C. Thao, E. V. Munson, T. N. Nguyen. Software Configuration Management for Product Derivation in Software Product Families. In ECBS'08, 2008.

[8] J. Gurp and C. Prehofer. Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families. Variability Mgmt., Workshop at SPLC'06.

[9] Subversion. http://subversion.tigris.org.

[10] X. Peng, W. Zhao, Y. Xue, Y. Wu. Ontology-Based Feature Modeling and Application-Oriented Tailoring. In ICSR'06, 2006.

[11] X. Peng, L. Shen, W. Zhao. Feature Implementation Modeling based Product Derivation in Software Product Line. In ICSR'08, 2008.