Synchronized Architecture Evolution in Software Product Line using Bidirectional Transformation

Liwei Shen, Xin Peng, Jiayi Zhu, Wenyun Zhao
School of Computer Science, Fudan University, Shanghai 200433, China
{shenliwei, pengxin, 072021130, wyzhao}@fudan.edu.cn

Abstract

In the long-term evolution of a Software Product Line (SPL), how to ensure the alignment between the reference and application architectures is a critical problem. Existing ad-hoc methods for architecture synchronization cannot ensure the completeness. In this paper, we propose a model-driven method for synchronized SPL architecture evolution using bidirectional transformation, a well-developed technique with solid mathematical foundation. Based on the model-based architecture representation, we capture the variability-intensive consistency relations between reference and application architectures and specify them with Beanbag [1], a declarative language supporting operation-based synchronization. Then, with the generated synchronizer and additional mechanisms, we can achieve coordinated architecture evolution through periodic synchronizations.

1. Introduction

Software product line (SPL) usually involves a long-term evolution with scope extensions or changes in existing applications after it is firstly established. In the evolution, how to ensure the alignment between the core assets and the applications while allowing different evolution directions is a big challenge [2].

In this paper, we focus on architecture evolution in SPL lifecycle. Architecture plays a central role in SPL development and evolution [2]. In an SPL, all the applications share a common reference architecture while each application architecture can be seen as a specific instance of it [3]. Usually, application architecture is first derived from the reference architecture with customization and extension. After that, the application architecture may be maintained independently to ensure accurate and responsive customer services [4]. Under some circumstances, the changes upon application architectures cannot be ignored in the SPL organization so that they should be merged back to the reference architecture as well as be propagated to other application architectures to ensure the consistency of the SPL assets. Manually evolution synchronization is possible, but is highly error-prone and labor-intensive [4]. Therefore, some kind of automatic mechanisms should be developed to help implement synchronized architecture evolution. Our previous work [2] proposed an architecture-based SPL evolution management method with an ad-hoc architecture merging algorithm. Chen et al. [4] also proposed their differencing and merging algorithms based on the xADL 2.0 framework [5]. However, these ad-hoc methods cannot ensure the completeness and are hard to extend.

In this paper, we propose a model-driven method for synchronized SPL architecture evolution using the technique of bidirectional transformation, which supports synchronization by propagating updates between two heterogeneous models [1]. Bidirectional transformation is useful for heterogeneous synchronization [9, 10]. Especially, Beanbag [1], as a new language for operation-based synchronization of data with inter- and intra-relations is adopted. We first represent both the reference architecture and application architecture with xADL 2.0 [5], an XML-based architecture description language. Then we identify both inter- and intra-model consistency relations, and specify them declaratively in a Beanbag program. Finally, the Beanbag program can be compiled into an architecture synchronizer, which takes initial architecture model and architecture updates as input and produces new updates to make the architectures consistent.

The remainder of this paper is organized as follows. Section 2 presents some background introductions. Section 3 introduces our method and Section 4 shows a working example adopting our method. Then we discuss related work in Section 5 and conclude the paper in Section 6.

2. Background

In this section, we present the background introductions to the synchronized architecture evolution in SPL and bidirectional transformation with Beanbag[1].

In the lifecycle of a SPL implementation, the initial version of an application architecture is usually derived from the reference architecture by customization and extension. After that, the reference architecture and the application architectures can evolve independently to meet their own evolution goals. Temporary deviations between the reference and application architectures are allowed, but periodic synchronizations should be performed to reunify the reference and application architectures if the changes are of great importance to the entire SPL organization as well as its clients. After synchronization, the reference and the application architectures can evolve independently again till the next synchronization. This evolution cycle provides the required flexibility for the application engineering to quickly respond to changing customer requirements, and ensure the consistency at the same time. Detailed introduction to synchronized architecture evolution process can be found in [2].

On the other hand, bidirectional transformation is an essential technique required in model-based development methods to maintain the consistency among different models or views. In SPL development, both the reference and application architectures can be formally modeled using xADL 2.0. Then based on the consistency relations, we can implement the synchronization between reference and application architectures with bidirectional transformation tools.

Beanbag [1] is a declarative language to support operation-based synchronization with inter- and intra-relations. It takes model updates (e.g. deleting a component) as input and produces new updates to be applied on the models to make them consistent. On the other hand, inter-relations indicate the dependency between elements in two replicas of data, while intra-relations indicate the dependency within one replica. In Beanbag these two types of relations are captured in a unified way, and are treated equally [1]. Moreover, Beanbag provides Java-based implementation that enables us to integrate the architecture synchronizer into our tool for SPL development and evolution [2]. Readers can refer to [1] or visit the Beanbag website for detailed introductions to Beanbag.

[1] The Beanbag Website: http://www.ipl.t.u-tokyo.ac.jp/~xiong/beanbag.html.

3. Our Method

A single-round process of our method is presented in Figure 1. We specify the intra- and inter-architecture consistency relations in Beanbag program, and then Beanbag compiler compiles the program and generates the architecture synchronizer. A Beanbag synchronizer has two procedures: ***initialize*** and ***synchronize***. The ***initialize*** procedure initializes the synchronizer with initial model input and produces new updates, while the ***synchronize*** procedure synchronizes models by taking user updates as inputs and producing new updates for consistency.

Based on the synchronizer, we can synchronize the architecture evolutions with the following process. First, the original reference architecture and the application architecture produced by customization are provided as inputs to the ***initialize*** procedure, which may output some initial synchronization updates. Second, both the updates to the reference architecture and the application architecture are provided as inputs to the ***synchronize*** procedure. Third, updates produced by the ***initialize*** as well as the ***synchronize*** procedure are provided by the architect as suggestions. These suggested updates, if accepted, will be applied to the modified reference architecture and application architecture. Some failures may also be reported in the ***initialize*** and the ***synchronize*** procedure if there are conflicts, which require further discussion and consideration.

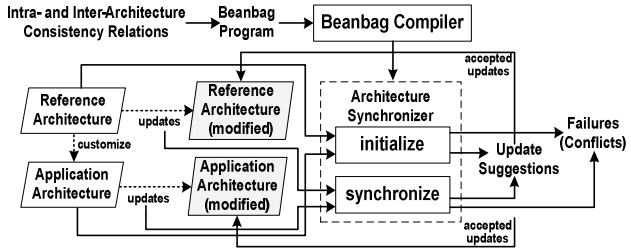

Fig. 1. Method overview

3.1 Model-based Architecture Representation

In Beanbag, each model element is denoted by an object, which consists of a set of attributes including a unique key, and each attribute of an object points to either a primitive value or a key of another object as a reference [1]. Therefore, we design the meta-model of architecture which involves four types of basic architectural elements: **Component**, **ComponentType**, **Connector** and **Link**. In the model, each architectural element is described by some attributes, including a unique key. Other properties can be of primitive types (e.g. **String**, **Boolean**) or

reference to other elements (can be represented by the key of other objects).

The basic structure of our model is consistent with xADL 2.0. **Component** has the attribute **optional** to denote whether it is optional, and the attribute **compType** referring to its type. **ComponentType** uses the attribute **vtype** to denote its variability type, i.e. normal, variable or abstract. Variable **ComponentType** has some normal **ComponentType** as its variants, while abstract **ComponentType** has no variants but is to be instantiated in applications. In order to represent the variant relation, normal **ComponentType** uses the attribute **variantOf** to refer to its parent **ComponentType** if it is a variant component type. **Connector** represents communication logic between components, and **Link** denotes the connection between a component interface and a connector interface.

In our method, both the reference architecture and application architectures share the same meta-model. The only difference is that in application architectures, all the variation points should have been resolved, so in application architectures there are no variable or abstract **ComponentType** and the **optional** attribute is meaningless. With the model-based representation, we can describe both reference and application architectures as a set of Beanbag objects. For example, we can represent the **Component** objects as the dictionary presented in Table 1. In the dictionary segment, a mandatory component ***ShoppingCart*** and an optional component ***TransInform*** are listed with their type information.

Table 1. An example dictionary for **Component** objects

1	{ShoppingCart->{compType->ShopCartType, key->ShoppingCart, optional->false},
2	TransInform->{compType->InformType, key->TransInform, optional->true}…}

3.2 Variability-intensive Consistency Relations

Based on the model-based representations, we can capture all the inter- and intra-consistency relations and specify them with Beanbag. The inter-relations are consistency constraints between reference architecture and application architecture under the variability constraints. The intra-relations are mostly come from the definitions of architectural elements in xADL 2.0. Besides, there are also some trivial consistency relations that can be directly checked and ensured in architecture editing, e.g. the interfaces of a component should be consistent with the specified signatures in the assigned component type.

In the next sub-sections, we will introduce the consistency relations for the elements in the architecture. The differences between a reference architecture and an application one according to the consistency relations indicate the reasonable customization or derivation in SPL, and are free of synchronization operations. However, the inconsistent relations in architecture evolution need to be distinguished since they are the main motivation towards synchronization and propagation for the consistency in architectures, similar to the "fixing procedure" in [8].

3.2.1 Consistency Relations for Components

According to the semantics of optionality, an optional component in the reference architecture can be bound or removed in the application architecture, and a mandatory component should be reserved.

The consistency relations for **Component** can be specified in Beanbag as Table 2 (simplified for reading). The `for` statement in line 3 declares that each pair of (`pComp, dComp`) from the application architecture and reference architecture with the same key should have one of the following consistency relations. The synchronization statements in line 4-8 consists of a series of OR formulas connected by "|" (disjunction), and each OR formula may have several AND relations connected by ";" (called conjunction).

Table 2. Consistency relations for **Component**

```
3  for [pComp,dComp] in [applicationComps, domainComps]{
4    {pComp=null; dComp."optional"="true"; dComp<>null;}
5  | {containmentRef<attr="key">(pComp, normalAppComps);
     dComp=pComp;}
6  | {containmentRef<attr="key">(pComp, newAppSpecificComps);
     dComp=null;}
7  |
   {dComp."name"=pComp."name";dComp."optional"=pComp."optional";
8    dComp."key"=pComp."key";}
   | {dComp=null;pComp=null;}}
```

For example, line 4 specifies the relations for removed optional component; line 6 specifies that for a component belonging to application-specific extensions (e.g. ***Comp3*** in Figure 2), corresponding domain component can be null (not exist). In the statements, the relation `containmentRef` is provided by the Beanbag standard library. And some secondary element sets (e.g. `newAppSpecificComps`) are used. Their members are defined by other relations.

Besides consistent variability customization, there are also inconsistent architecture evolutions to be synchronized as following:

- A mandatory component does not exist in the application architecture. In this case, the

component should be turned to be optional in the reference architecture.

- New component is added in the application architecture. In this case, we should first determine whether the new component is a part of the application extension. Figure 2 shows a contrastive example of inconsistent structure evolution and application extension on structure. In the example, ***Comp3*** is a consistent application extension, since it serves for the application-specific implementation of ***Comp2*** only. Another new component ***Comp4*** is connected to domain component ***Comp1***, so it is an inconsistent structure evolution. For consistent application extension, we can leave the new components as application-specific part. For inconsistent new component like ***Comp4***, we should add it to the reference architecture as an optional component.

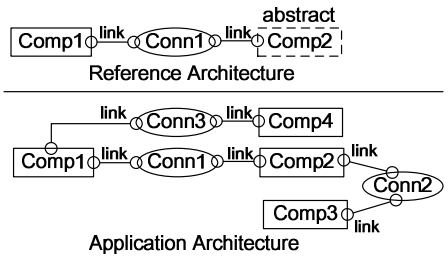

Fig. 2. Inconsistent structure evolution and application extension on structure

3.2.2 Consistency Relations for Component Types

For those consistent component structures, we can further consider the consistency on component types according to the three variability types (normal, variable and abstract). For a component C with the type CType in the reference architecture, assume corresponding component in the application architecture has the type CType', we have the following situations. (1) if CType is normal component type, then CType' should be the same as CType. (2) if CType is variable component type, then CType' should be any of the variant component types of CType (3) if CType is abstract component type, then CType' can be seen as the application-specific instantiation to CType. Besides these consistent variability customizations, there are also inconsistent type evolutions as shown in Table 3. In the first case, a new variable component type will be introduced to the reference architecture. In the latter case, a new variant will be added to an existing variable component type.

3.2.3 Consistency Relations for Connectors and Links

Architectural variability in our method is embodied in components and component types, so we have no variability-related consistency rules specific to connectors and links. The role of connectors and links in the architecture synchronization is twofold. First, they provide the necessary context information to determine the evolution types of components. For example, in Figure 2, Comp3 is determined to be part of the application extension based on the link and connector related to it. Second, they are involved in some accompanying consistency relations for components. For example, the require part and the server part of a link cannot be null. When an optional component is removed in the application architecture, corresponding links should also be deleted.

Table 3. Synchronization policies for inconsistent type evolutions

Variability Type	Situation	Synchronization Policy
Ctype is Normal	***CType'*** is different from ***CType***	Assign a new variable component type to ***C*** in the reference architecture with ***CType*** and ***CType'*** as the two variants
Ctype is Variable	***CType'*** is different from any of its prescribed variant component types of ***CType***	Add ***CType'*** as a new variant of ***CType***

3.3 Multiple Options Resolution

In some cases, there may be multiple options, i.e. several alternative updates for the same synchronization problem. In Beanbag, multiple options can be represented by disjunction (|) relations and the order denotes priority, i.e., the synchronizer will first try to generate updates to meet the first relation, if failed then the second, the third, till the last one. However, in architecture synchronization, it is often up to the domain or application architect to make the decision. For example, when the reference architecture involves a new optional component, we can either add it to the application architecture or ignore the consistency relations on components. Our solution is first labeling to-be-determined architectural elements in synchronization and then request decisions from the architect. For example, in the above case, the new component can be propagated to the application architecture with the property "optional=true" to indicate it is to-be-determined. Then the SPL tool can detect this kind of pending points and request decisions from the architect. After that, new updates (e.g. remove

the component in the application architecture) can be sent to the synchronizer to start a new synchronization round.

3.4 Synchronization Conflicts

In some cases, the synchronizer may report failures if no consistent updates can be found to satisfy all the consistent relations. It is conflicts that mean inconsistent architecture decisions which cannot be synchronized. Figure 3 shows an example of conflicting architecture evolutions. In this example, ***Comp3*** connects to both the application-specific implementation of ***Comp2*** and a domain component ***Comp1***, so the synchronization rules for both application-specific extension and new component to be added are applicable for it. In beanbag, when two updates try to change the same location to different values, there is a conflict and a failure will be reported [1]. For conflicts, the domain and application architects should do further discussion and negotiation to obtain consistent decisions. If resolved, the synchronization can be restarted with the new consistent update decisions. Besides these inherent conflicts, rejecting suggested synchronizing updates usually means to bring conflicts, e.g. rejecting to turn a domain component that is removed in the application architecture to be optional.

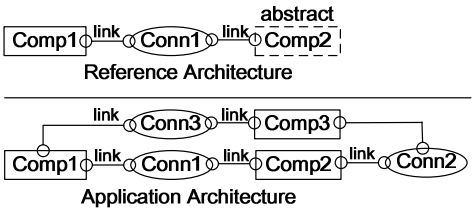

Fig. 3. An example of conflicting architecture evolutions

4. Working Example

In this section, we demonstrate our method with a simplified online-shopping product line. Figure 4 presents the reference architecture and an evolved application architecture, in which the component type assigned to each component is represented as a gray pane under the component. Besides, components with dashed lines denote optional components, component types with dashed lines are variable or abstract. Variants for variable component types are listed in corresponding panes. Besides variability customization, e.g. removing optional component ***VirtualAccountMgt***, the application architecture also involves several inconsistent adaptations:

1) instead of choosing a prescribed variant component type for ***Payment***, a new variant component type ***PayByCashType*** is introduced;
2) a new component ***CashConfirm*** is introduced with a new component type;
3) a new component ***AfterService*** is introduced with a new component type;
4) a new component ***CustomerAnalysis*** is introduced together with the application-specific implementation ***MyDiscountType*** for the abstract ***DiscountType***.

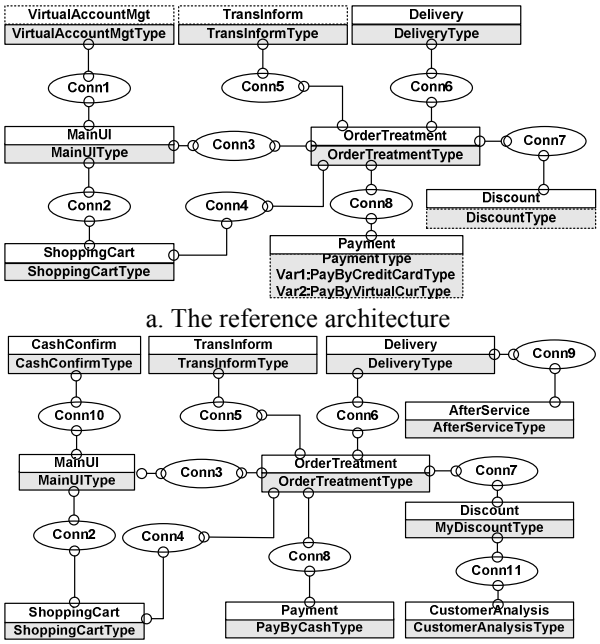

Fig. 4. Architectures before synchronization

In the synchronization, we generate the dictionary-based representations for the architectures according to the meta-model mentioned in 3.1, and provide them as input to the synchronizer. After applying the output updates on the original architectures, we can obtain the new reference architecture as shown in Figure 5, which involves the following updates:

1) ***PayByCashType*** is added as a new variant for ***PaymentType***;
2) ***CashConfirm*** is added as an optional component;
3) ***AfterService*** is added as an optional component.

The new application component ***CustomerAnalysis*** is not added into the reference architecture, since it is a part of the extensions for the abstract component ***Discount***.

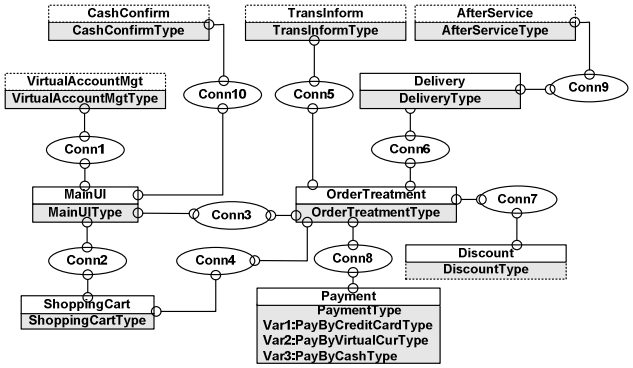

Fig. 5. The reference architectures after synchronization[2]

5. Related Work

In the area of software evolution and maintenance, there has been a lot of work on architecture evolution. However, very little work has been conducted on architecture evolution in SPL development, especially evolution synchronization among architectures. The xADL group presents their evolution management tool for SPL architectures in [6], which supports versioning of architectural elements. They also propose an architecture differencing and merging method with ad-hoc algorithms in [4]. In their algorithms, there are no considerations for variability abstract and the difference between variability customization and inconsistent adaptations. Recently, some researchers introduce model synchronization techniques into evolution management. Ivkovic et al. [7] use model transformations to synchronize software artifacts at different levels of abstraction such as architecture diagrams, object models, and abstract syntax trees, etc. In our method, we focus on synchronized evolution between the reference architecture and application architectures in an SPL, based on the variability semantics in the variability-intensive architecture description language.

6. Conclusion and Future Work

In SPL development and evolution, both the reference architecture and the application architectures evolve with different goals and disciplines [2]. In order to ensure the consistency among the reference and the application architectures, evolution synchronization must be considered. In this paper, we have shown that bidirectional transformation techniques can be applied with model-based architecture representations to achieve effective and synchronized architecture evolution in SPL development. Furthermore, we can use our SPL development tool to bridge the automated synchronization and subjective architecture decisions. In the future work, we will further integrate the synchronizer with our SPL tool, to integrate architecture synchronization with architecture versioning and to unify the evolutions of other artifacts such as components under the architecture evolution synchronization.

Acknowledgments. We thank Yingfei Xiong at University of Tokyo and Prof. Zhenjiang Hu at the Japan National Institute of Informatics for their help and support in Beanbag. This work is supported by National Natural Science Foundation of China under Grant No. 60703092 and No. 90818009, and National High Technology Development 863 Program of China under Grant No. 2007AA01Z125 and 2009AA010307.

References

[1] Y. Xiong, Z. Hu, H. Zhao, M. Takeichi, H. Song, H. Mei. Beanbag: Operation-based Synchronization with Intra-Relations. Grace Technical Reports, GRACE-TR-2008–04, National Institute of Informatics, Japan, December 2008.
[2] X. Peng, L. Shen, W. Zhao. An Architecture-based Evolution Management Method for Software Product Line. In SEKE 2009.
[3] K. Pohl, G. B¨ockle, and F. Linden. Software Product Line Engineering: Foundations, Principles and Techniques. Springer, September 2005.
[4] P. Chen, M. Critchlow, A. Garg, et al.. Differencing and Merging within an Evolving Product Line Architecture. In PFE'03, 2003.
[5] E. M. Dashofy, A. Hoek, R. N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM TOSEM, 2005, 14(2).
[6] A. Garg, M. Critchlow, P. Chen, et al.. An Environment for Managing Evolving Product Line Architectures. In ICSM'03, 2003.
[7] I. Ivkovic, K. Kontogiannis. Tracing Evolution Changes of Software Artifacts through Model Synchronization. In ICSM'04, 2004.
[8] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, H. Mei. Supporting Automatic Model Inconsistency Fixing. In ESEC-FSE'2009.
[9] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In Proc. 10th MoDELS, pages 1–15, 2007.
[10] J. Foster, B. Greenwald, T. Moore, C. Pierce, A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst., 29(3):17, 2007.

[2] The complete Beanbag program for the architecture synchronizer and the input/output updates in the case study can be found on: www.se.fudan.edu.cn/SPL/beanbag.html.