Making Use Case Slices Manage Variability in Aspect-based Product Line

Satish Mahadevan Srinivasan and Mansour Zand

University of Nebraska at Omaha, 68132 Omaha, Nebraska, U.S.A.

Abstract. Use case slice, clearly lacks composition mechanism due to which it is difficult to manage variabilities in Software Product Lines. A use case slice can only convey the design of a single member of a product line. Aspect-based modeling of use case slices look to be a promising solution but there are few issues within their composition mechanism. Aspect-based modeling of use case slices clearly lack a strong and familiar algebraic model and also fails to address precedence management issues among artifacts such as pointcuts and advices. This paper suggests a composition mechanism, for aspect-based implementation of use case slices, which would provide a familiar algebraic model and will resolve issues related to precedence management. In this paper we have discussed about a hypothetical aspect-based Expressions Product Line (EPL) and have shown how the use case slices can be used to model the variabilities in EPL.

1 Introduction

Product line engineering is defined as "a set of software-intensive systems that shares a functionally overlapping, managed set of features that satisfy the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way" [2]. The key benefits of switching to product line engineering are two folds, firstly, it tries to improve the development efficiency for the family of related systems by facilitating large scale reuse and secondly, it allows sharing the maintenance effort for developing other members of the product line [1]. Product line requirements are of two types namely commonality and variability. The variability deals with the separation of the generic from specific aspect or concern in a product line infrastructure. If the variability issues are not addressed properly the system may fail to deliver what it promises. Thus there is a need for a modeling language with a composition mechanism that can efficiently capture the variabilities in a product line system and ensures that the system functions as it is intended to.

Section 2 below gives a brief introduction to the aspect-based approaches in software engineering and discusses about the modularizing units of the use case, use case slices. Section 3 gives a brief introduction to the aspect-based EPL product line. This paper tries to demonstrate how the aspect-based use case slices can help in managing variabilities or concerns across different products in the EPL product line. The demonstration part constitutes section 4. Section 5 finally concludes the paper.

Mahadevan Srinivasan S. and Zand M. (2009).
Making Use Case Slices Manage Variability in Aspect-based Product Line.
In *Proceedings of the 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 141-148
DOI: 10.5220/0002200301410148

2 Aspect-based Implementation of Use Cases

Aspect-based approaches provide a more advanced means for modeling features through the concept of separation of concerns. The separation of concern is established by modularizing the crosscutting concerns (variabilities) as a separate software artifact called aspects. This mechanism clearly separates the distinction between communality and variability (concern) thus supporting evolution, reusability, improving traceability and enabling consistency checking [1]. The separation of concerns can be realized using the Aspect-Oriented Programming (AOP) where each concerns can be modeled as aspects [5].

The use case is a common modeling practice. Use case is a technique to capture the system functionality and requirements using the UML [4]. According to Jacobson and Ng [3] the implementation of use cases using the traditional object oriented languages and technique typically breaks the modularity of use case. When the use cases are implemented using an Object-Oriented languages they may lose their modularity as the resultant code could be scattered and tangled across different modules [4]. Thus in [3] Jacobson and Ng have proposed a aspect-based modeling of use case or use case slice, which are modularized units of the use case.

A use case slice can be modeled as a special kind of package stereotyped as <<use case slice>> consisting of a collaboration (UML diagram describing the realization of a use case (interaction, class etc.)), specific classes; which are specific to a use case realization or commonality or base code, and specific extensions or aspect; extensions to existing classes specific to a use case realization or concerns or variabilities [4]. In this paper we will be assuming that a use case slice will have a name, a collaboration symbol (a dashed ellipse), the base class (specific classes) and concern (specific extension). The concern in the use case slice has two specific compartments, one for listing the pointcuts and the other to list the advices. In the next section we will pictorially show how a use case slice implementation of a particular product line feature in our EPL looks like.

According to Lopez and Batory in [4] use case slices do not provide modeling support for the variability or concerns entailed by a product line. Thus a use case diagram conveys the design of a single member of a product line. The drawbacks observed by them are:

1. The use case slices lack a clear composition model to map use case models to concrete working implementations.
2. The translation of the use case slices to the source code in AspectJ highlights the missing of an important compositional issue, precedence management.
3. The concern's (specific extensions) being specific to a use case slice inhibits its reuse. This can be mitigated by keeping the concerns not so specific to a use case slice.
4. A single use case is not limited to have a single concern (pointcut and advice) and so a single use case slice can have multiple concerns [3]. For example, if multiple advices are superimposed at the same joinpoint then there is no way to assign an ordering on the composition of these advices at a given joinpoint. In short the composition mechanism for the aspect-based implementation of use case slices has to address issues related to precedence management.

In the next section we will be discuss about a aspect-based product line EPL and also show how the use case slices can be used to model the features of the EPL.

3 EPL Aspect-based Product Line

This paper uses a simple aspect-based product line based on the Extensibility problem called as Extensibility Problem Line (EPL) discussed in [4]. The EPL consists of a family of programs with a mix of new operations and datatypes to represent expressions of the language defined below:

EXP :: = Lit | Add | Neg

Lit :: = <non-negative integers>

Add :: = EXP "+" EXP

Sub :: = EXP "-" EXP

On this grammar two types of operation can be defined:

1. Print: This operation displays the string value of the expression. The expression 3+2 is represented as a three-node tree with a Add node as the root node and two Lit nodes as leaves. The operation Print applied to this node displays the string "3+2".
2. Eval: This operation evaluates an expression and returns a numeric value rounded up to two decimal places. Applying the operation Eval to the tree of the expression 3+2 yields 5 as the result.

The EPL has been chosen in this context because this problem has been studied widely within the context of programming language design where the focus is in archiving data type and operation extensibility in a type-safe manner. The EPL is a two-dimensional matrix where the rows represent the data types and the columns specify the operations. Each of the matrix entry is a feature or a program of this aspect-based product line. Each of these features or the programs implements the operation, described by the column, on the data type, specified by the row. Figure 1 below gives a matrix representation of the EPL. The operation Eval and Print on data types Lit and Add is composed with features lp, le, ap, ae, sp and se.

A class named operation creates instances of the datatype classes and invokes their operation. In the class operation there are two functions namely operate1() and operate2(). The operate1() takes two argument of type Exp and operates on them i.e. "Exp2 + Exp1" or "Exp2 – Exp1". The operate2() takes a single argument of type Exp and operates on them i.e. "Add: Exp→ Exp + Exp" or " Sub: Exp→ Exp - Exp" [4]. Figure 2 below shows how particular features ap of the EPL can be represented using a use case slice.

In Figure 2 above it can be observed that the concern is an integral part of the use case slice ap. It is because of this integration (concerns being integral part of the use case slice) the reuse of the concerns gets inhibited. To mitigate this inhibition we have proposed to keep concerns away from the use case slice (base code) thus enabling the concerns to be reusable software artifacts. In addition to that we propose

	Print			Eval		
Lit	***Exp*** *Void Print() lp*	***Lit*** *Int value Lit(int) Void Print()*	***operation*** *Lit ltree Operate1() Operate2()*	***ΔExp*** *Int eval() le*	***ΔLit*** *Int eval()*	***Δoperation*** *Δoperate2()*
Add	***ap***	***Add*** *Exp left Exp right Add(Exp, Exp) Void Print()*	***Δoperation*** *Add atree Δoperate1() Δoperate2()*	*ae*	***ΔAdd*** *Int eval()*	***Δoperation*** *Δoperate2()*
Sub	***sp***	***Sub*** *Exp left Exp right Sub(Exp, Exp) Void Print()*	***Δoperation*** *Sub stree Δoperate1() Δoperate2()*	*se*	***ΔSub*** *Int eval()*	***Δoperation*** *Δoperate2()*

Fig. 1. Matrix representation of EPL.

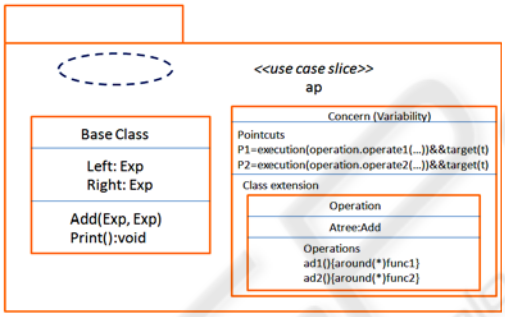

Fig. 2. Use Case Slice for feature ap.

to breakdown the concern's in to pointcuts and advices and make each individual piece as a reusable software artifacts. Finally we have proposed a composition mechanism that will take care of how these software artifacts will bind (binding of advices with pointcuts) together during run time.

The advices can selectively bind with the pointcuts using a hash table lookup and the stereotype <<binds>> as shown in Figure 3 below. Each row entry in the hash table lookup holds a pair suggesting which advice will bind with which pointcut for a particular combination of the base class and pointcut. For example a pair (P1,ad1()) suggests that the advice ad1() will bind with pointcut P1. The blue boxes in Figure 3 below represent the hash table lookup entries. The hashing key for selecting a particular row in the hash table lookup is determined by the combination of the base class (base code) name and the name of the pointcut. The hash table lookup thus has two columns, the first column holds the hashing key value (base class name, name of the pointcut) and the second column holds a set of selective binding suggestions (pointcut name, advice name). Therefore our composition mechanism supports a predetermined selective binding between the advices and the pointcuts. These selective binding criteria's are hard coded and they determine which particular advice will bind with which particular pointcut. The design of the hash table lookup and the issues behind its design are beyond the scope of this paper.

Though separating the pointcuts and advices from the use case slice enhance their reuse but at the design stage (defining relations between the artifacts) these separated

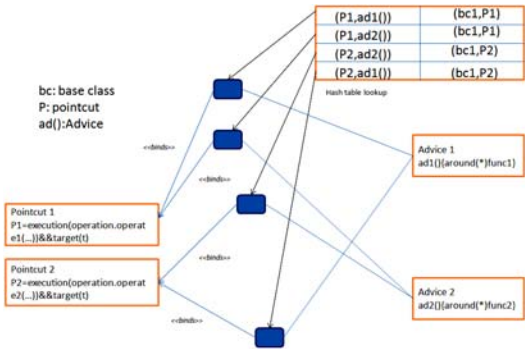

Fig. 3. Binding diagram between pointcuts and advices using the hash table lookup and the <<binds>> stereotype.

software artifacts may sometimes lead to potential conflicts arising due to their interaction. A situation, in which many different advices may be required to be superimposed at a particular joinpoint, will result in to a potential inconsistency problem [6]. According to Zhang et. al. incorrect interference of the advices at a particular joinpoint may lead to the change of the state of the base program. This problem can be mitigated by declaring precedence relationship which tries to order the composing of the advices at a particular joinpoint. Thus it will be the responsibility of our composition mechanism to assign precedence relation on selective binding of advices with pointcuts. Thus it is important to ensure that our composition mechanism implement solutions to address precedence management issues and also support an algebraic-based composition model. The algebraic-based composition is suggested in the belief that reusability of pointcuts can be archived and at the same time precedence logic can also be formulated.

4 Proposed Composition Mechanism

To develop a composition mechanism for composing the advices and pointcuts we have to look in to the following issues:

1. To provide an algebraic model for composing the pointcuts.
2. To introduce a precedence declaration on and between advices.

To enhance the reusability of the pointcuts and advices the composition mechanism has to implement two distinct categories of composition mechanism namely the pointcut composition and advice composition. This composition mechanism is similar to the Motorola weaver Weavr discussed in [6].

4.1 Pointcut Composition

The algebraic model for our composition mechanism is inspired by the algebraic model discussed in [6]. To support the composition of the pointcuts the composition

mechanism should support boolean operators like && (AND), || (OR) and the! (NOT). The AND operator indicates the intersection of the set of joinpoints. The OR operator indicates the union of the set of joinpoints and the NOT operator indicates the difference of the set of joinpoints. These boolean operators help in composing a new pointcut using the existing pointcuts. For example, if there is an expression in our EPL such as

```
Add(Add(Exp1), Add(Exp2))where s(Exp exp1,Exp  exp2)
{operate2(exp2), operate2(exp1), operate1(exp1,exp2)}.
```

Let us suppose we need a pointcut P3 for this expression that combines both pointcuts P1 and P2 in such a way that if one pointcut exists then the other should not. This can be represented in our composition mechanism as (((P2 → !(P1)) && (P2 → !(P1))), thus avoiding the creation of the pointcut P3 and enhancing the reuse of the pointcut P1 and P2. Here P1, P2 and P3 are the names of the pointcut.

Combining the basic operators AND, OR and NOT much complex boolean operators namely, XOR and NAND can be implemented in our composition mechanism. We are currently investigating on the implementation complexity and benefits offered by the Boolean operators XOR and NAND. Considering the limited scope in terms of space in this paper we have avoided further discussions about XOR and NAND Boolean operators.

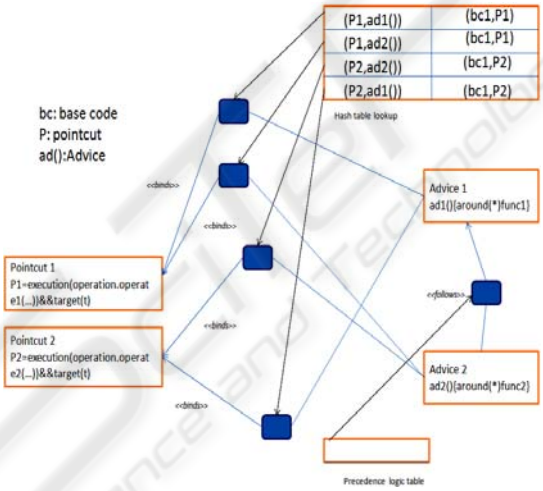

Fig. 4. Composing advices using the precedence logic table and with the stereotype <<follows>>.

4.2 Advice Composition

The advice composition tries to bind and execute the advice instance at a given joinpoint in a particular order. It was pointed out earlier in this paper that sometimes different advices may have to bind at a particular joinpoint. In such circumstances it is important to ensure that the order in which the advices bind at a particular joinpoint is controlled by defining some precedence ordering. In this paper we suggest using

the stereotype <<follows>> to impose a precedence order between different advices. Figure 4 above pictorially represents the composition of two advices, advice 1 and advice 2, using the precedence logic, given in the precedence logic table and the <<follows>> stereotype. The precedence logic table is also a hashing table and is similar to the hash table lookup discussed in the earlier section. The precedence logic table suggests the precedence relation (order of execution) between any two or more advices that will bind at a particular joinpoint. The hashing key for the precedence table is a set that lists the name of the all the advices that will bind at a particular joinpoint. The design of the precedence logic table and the issues behind its design is beyond the scope of this paper. A small example is consider here to demonstrate how our proposed composition mechanism will tackle issues related with precedence management at a particular joinpoint if Advice 1 has precedence over the Advice 2, and if Advice 1 and Advice 2 has a "before" and "after" actions, then the "before" of Advice 1 will be executed before the execution of the "before" of Advice 2 and the "after" of Advice 2 will be executed before the "after" of Advice 1. If the pointcuts P1 and P2 match a single joinpoint in the base code, then the advice instantiation order at this joinpoint would be Pointcut1-Advice1, Pointcut2-Advice1, Pointcut1-Advice1, and Pointcut2-Advice2. If both the Advice1 and Advice2 have a "before" and "after" action then the execution order enforced by our composition mechanism would be:

- Before action in the Pointcut1-Advice1.
- Before action in the Pointcut2-Advice1.
- Before action in the Pointcut1-Advice2.
- Before action in the Pointcut2-Advice2.
- Original actions to be performed at the joinpoint.
- After action in the Pointcut2-Advice2.
- After action in the Pointcut1-Advice2.
- After action in the Pointcut2-Advice1.
- After action in the Pointcut1-Advice1.

The advantages realized using this advice composition is:

1. The advices being an independent entity can selectively bind with any point-cuts using the <<binds>> stereotype based on the hard coded selective binding criteria thus loosely coupling the software artifacts and enabling them to be reusable.
2. If two or more advices are to be associated at a single joinpoint then a precedence management can be established between these advices using the <<follows>> stereotype and the precedence logic hard coded in the precedence logic table.

The crux behind this approach is to first develop a use case, based on the requirement obtained from the stakeholders of the system and convert those use cases in to use case slices. After the use case slices are developed the concern (pointcuts and advices) from the use case slice is removed and is maintained as separate features or software artifacts. Using the composition mechanism, discussed above the use case slices can be exploited to model variabilities entitled by the family of related products in the aspect-based product line systems.

5 Conclusions and Contribution

In this paper we have shown how the use case slices can be used for modeling and synthesizing the products of the aspect-based product line system EPL. The main contributions of this paper are A) Proposing that keeping software artifacts, pointcuts and advice, separate from the use case slice would enhance reuse of the software artifacts and as well enable use case slices to support the variability encountered in the aspect-based product line systems. B) Proposing a composition mechanism support for use case slices in the form of pointcuts composition and advice composition. This composition mechanism provides an algebraic model to compose various software artifacts and at the same time helps resolve issues related to precedence management.

References

1. Bayer, J.: Separating Concerns in Product Line Engineering. Fraunhofer Institute for Experimental Software Engineering (IESE). (2001)
2. Clements, P.C., Northrop, L.: Software Product Line: Practices and Patterns. Addison-Wesley, (2001)
3. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2nd edition, (2004)
4. Lopez-Herrejon, R.E., Batory, D.: Modeling Features in Aspect-Based Product Lines with Use Case Slices: An Exploratory Case Study, ftp://ftp.cs.utexas.edu/pub/predator/LopezHerrejon-Batory.pdf
5. Siy, H., Aryal, P., Winter, V., and Zand, M.: Aspectual Support for Specifying Requirements in Software Product Lines. In: Proceedings of the International Conference of Software Engineering (ICSE 2007). Early Aspects Workshop. (2007)
6. Zhang, J., Cottenier, T., Berg, A.V.D., Gray, J.: Aspect Composition in the Motorola Aspect-oriented Modeling Weaver. Journal of Object Technology, vol. 6, no. 7, Special issue: Aspect-oriented Modeling (2007)