Feature Implementation Modeling Based Product Derivation in Software Product Line

Xin Peng, Liwei Shen, and Wenyun Zhao

Computer Science and Engineering Department, Fudan University, Shanghai 200433, China
{pengxin,061021062,wyzhao}@fudan.edu.cn

Abstract. Although there has been significant research spent on feature modeling and application-oriented customization and some effective methods have been proposed, product derivation in SPL (Software Product Line) development is still a time- and effort-consuming activity due to the complicated mapping between feature model and program implementation. In this paper, we propose a feature implementation modeling based method for product derivation. In the method, feature implementation model is introduced as the intermediate level between feature model and program implementation. The feature implementation model captures feature interactions (including cross-cutting interactions) in the finer role level, and help to clarify the complex mapping between feature and program implementation. So, feature-driven program-level customization and configuration can be enabled by the model and role instantiation. AOP (Aspect-Oriented Programming) is adopted as the implementation technology for product derivation on the program level. Then program-level composition can be implemented by aspect weaving to finally achieve the feature-driven product derivation.

1 Introduction

A fundamental reason for investing in SPL is to minimize the costs of product derivation [1]. The ideal mode of product derivation is constructing the final product by configuring and tailoring of core assets, following a prescribed process, and complemented by application-specific implementation of some parts. Feature-oriented domain requirements modeling points out a possible way to implement the customization-based requirement reuse [2]. So, most methods on feature modeling support application-oriented customization with constraint dependencies (e.g. [2][3]). However, the big gap between problem domain and solution domain make it difficult to map customization and tailoring on feature model to implementation program level.

The relation between problem and solution space is a many-to-many relation ([2][4][5]). Some intermediate mechanisms between features and components are proposed to improve feature-based architecture design and product derivation, e.g. responsibilities in [2] and component role in [4]. On the other hand, AOP (Aspect-Oriented Programming) as SPL implementation technology has attracted much attention (e.g. [6][7]) due to its enhancement on crosscutting features, adaptability and configurability. These concepts can help understand the reality of feature based design and implementation and provide guidance in practice. However, we still need systematic and practical methods for feature oriented product derivation, which should provide some automatic mechanism and tool support.

In this paper, we propose a feature implementation modeling based product derivation method for SPL development. Feature implementation model, an intermediate level between feature and program implementation, is introduced to link feature variability and program variability. In the model, each feature is logically implemented by some roles and interactions between roles are also modeled. Roles are instantiated by elements in the base programs or variability-related programs. AOP is adopted to implement the program-level composition for role interactions.

The remainder of this paper is organized as follows. Section 2 analyzes the problem of product derivation in SPL development. Feature implementation model and product derivation are introduced in section 3 and 4 respectively. Then a case study and tool support for the method are presented in section 5. Related works are discussed in section 6. Finally, we draw our conclusions with discussion in section 7.

2 Problem of Product Derivation in Software Product Line

In this section, we will analyze problems in product derivation with a simplified example of library management domain. The feature model is showed in figure 1 according to the ontology-based meta-model proposed in our previous work [3], in which decomposition relations are presented and lines with hollow circle represent optional elements. In the system, *BookAdd*, *Search&Browse*, *BorrowBook* and *ReturnBook* are basic functions. *BookPicShow* (show the book picture when browsing), *BorrowControl* (control book borrowing by prescribed policy) and *BookLog* (log when a book is added, borrowed or returned) are optional functions. Even if all the features are implemented, we will still find it hard to derive customized products from these core assets. Usually we will implement some business classes and several visual forms for book adding, browsing, borrowing and returning. Then we can see the bound of *BookPicShow* usually needs both an image container and a code segment of image data fetching to be added. Furthermore, there is interaction between *Search&Browse* and *BookPicShow*: *BookPicShow* should be activated when a book is searched and showed in the browsing form.

The interaction can be more complicated. The optional *BorrowControl* can even change the execution of base programs: if *BorrowControl* is bound, it will interrupt the execution of *BorrowBook* if it doesn't meet prescribed policy. Furthermore, feature interaction can even affects multiple features, e.g. *BookLog* affects *BookAdd*, *BorrowBook* and *ReturnBook*. Besides feature-level variability, there is also variability on design and program level, e.g. fetching image data from database or file system if *BookPicShow* is bound. This kind of variability is not visible on the feature level, but it will affect the product derivation also.

From this case, we can see problems in product derivation include: feature implementation scatters and bound of a feature may need adaptations on multiple program units; complex interactions between features and even crosscutting feature interactions; variability on multiple levels (requirement, design, etc). The root of the problem is the complicated mapping between feature and program implementation. In our method, role based feature implementation model is introduced to improve the situation by clarifying the mapping and feature interactions.

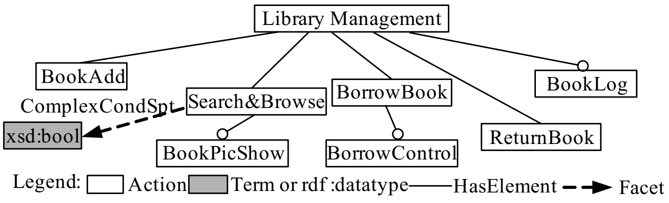


Fig. 1. Feature model of simplified library management domain

3 Feature Implementation Model

Feature implementation model is the logical design model for the implementation of features. It specifies all the necessary implementation roles for each feature and instantiates roles to program elements (i.e. class, method, etc), so as to map feature-level customization to program-level implementation. An SPL consists of a base implementation (mandatory features) and a number of variability-related features, and a product can be derived by selecting an arbitrary number of these features and combining with the base implementation [4]. In our feature implementation model, there are also base roles and variability-related roles. The latter are to be selected and configured to implement bound variability-related features, while the former provide linking points for variability-related programs to be composed into the product.

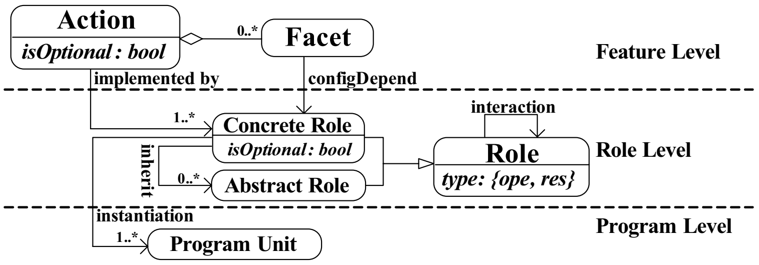


Fig. 2. Meta-model of the feature implementation model

3.1 Feature Implementation Meta-model

Meta-model of the feature implementation model is presented in figure 2, which is extended from our previous ontology-based feature model [3]. In the feature model, *Action*, representing business operations, is the basic element. *Facet* is introduced to provide more business details for actions, which can be construed as perspectives, viewpoints, or dimensions of precise description for Action [3]. Each action is implemented by one or more roles. A role is a logical unit, a responsibility which should be taken by program fragment for feature implementation. The concept of role here is similar to the responsibility in [2] and the role in [4]. Two kinds of roles are distinguished, one is operational role (*type=ope*), the other is resource role (*type=res*). An operational role is a functional segment, e.g. fetching image data for show. Resource role represents specific internal or external entity necessary for the implementation of a feature, e.g. an image container to show picture. Resource role in our method is similar to resource container proposed in [2], which can passively accept features' requests for resource storing, querying, and retrieving, and play the role of a medium of interaction between features.

A series of roles can interact with each other to implement a feature (mostly *Action* in our model) together. For example, from figure 3 we can see *BookPicShow* can be implemented by three roles. *ImageContainer* is an image container for picture show, *ImageFetch* is to fetch image data from database or file system, and *PicShowControl* takes the responsibility of controlling image fetching and showing. There are also interactions between roles from different features, e.g. *PicShowControl* is activated by role from *Search&Browse* to fetch and show the picture. Then the feature *BookPicShow* can be implemented by role interactions within it and across the role boundary. Role interaction clarifies how a user-visible feature is implemented by several logical sides and guides the program-level customization and composition. In our method, five kinds of interactions are identified as in table 1, in which interaction point denotes a role activates another role before or after execution of itself.

Table 1. Interactions between roles

Interaction	Description	Interaction Point
Involve	An operational role activates another operational role in a synchronous mode and makes it a part of the host operation.	**Before** or **After**
Inform	An operational role informs another operational role to activate in an asynchronous mode.	**Before** or **After**
Determine	Execution result of an operational role can determine the execution of another operational role, including whether execute or not and choosing a variant from several choices.	**N/A**
Access	An operational role reads or writes a resource role, or both, to fulfill its responsibility in specific feature implementation.	**N/A**
Introduce	A resource role is introduced into implementation unit of another operational role to be a sub-element.	**N/A**

Among the five kinds of interactions, the first three are between two operational roles, the last two between an operational role and a resource role. All these five kinds of interactions are embodied in figure 3. For example, *ImageFetch* is involved in *PicShowControl*, *BookChange* will inform *BookLog* to activate, *BorrowControl* (if bound) will determine the execution of *BorrowBook*, *PicShowControl* will access *ImageContainer* to show the image, and *ImageContainer* (if bound) will be introduced as an element of *BookSearch*. Interactions can occur between roles from different features, e.g. the interaction between *BookSearch* and *PicShowControl* in figure 3. Inter-feature role interactions are the embodiment of feature interactions and clarify the interactions in a finer granularity. It should be emphasized that not all the modeled role interactions will appear in a final product, since some roles reside in optional or variable features, e.g. *Determine* relation between *BorrowControl* and *BorrowBook* will not take effect if the feature *BorrowControl* is not bound. However, this interaction can help to compose the behaviors of *BorrowControl* and *BorrowBook* in the right way once *BorrowControl* is bound.

In some cases, role interactions will occur between a role and a set of roles with common characteristics. These interactions usually crosscut multiple parts of a system. For example, *BookLog* will be informed to activate by all the changes on books, e.g. *BorrowBook*, *ReturnBook*, etc. So, abstract role is introduced to denote a class of roles and provide expressions for crosscutting interactions. Abstract role does not reside in any feature and will be inherited by other concrete roles. For example, in figure 3, if the optional *BookLog* is bound, it will be attached to both *BorrowBook* and *ReturnBook*. An abstract role also has its role type (operational role or resource role), and role inheritance can only occur between roles with the same role type.

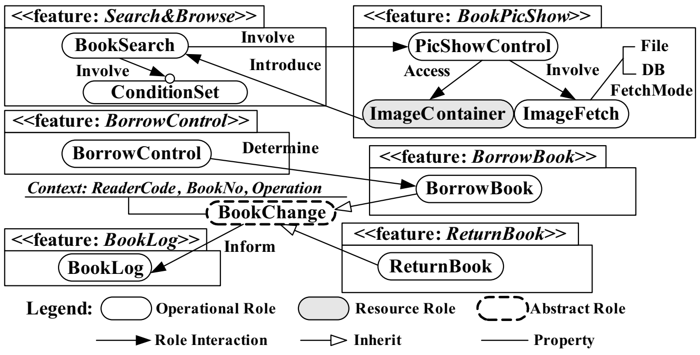


Fig. 3. Segment of feature implementation model for the library management domain

3.2 Variability in Feature Implementation Model

Role-level implementation for optional features is plain: roles for an optional feature are involved in the system or not according to whether the feature is bound. Similarly, as for specialization, each variant feature has its own role design and related roles are involved or not according to whether the variant is bound. In these cases, no additional variability should be considered. Our feature model provides the mechanism of partial variability for features, i.e. variable facet-value in our feature model, e.g. the facet *ComplexCondSpt* defined on the feature *Search&Browse* in figure 1. In this case, role-level variability should be modeled to support it. On the other hand, feature implementation model can also introduce new design-level variations for different implementation choices. For example, *ImageFetch* contains a design-level variation of fetching mode, i.e. read the image data from DB or file system. In our method, role-level variability is supported by optional roles and role properties. Optional roles should be further evaluated to be bound or removed even if the feature it resides in is bound. Role properties provide a kind of partial variability for roles, e.g. the property *FetchMode*. Different property-values mean different implementation modes in program level, e.g. different instantiations or parameters.

As mentioned above, some role-level variations are related to feature-level, others are completely design-level considerations. Example of the former is the optional role *ConditionSet*, which is designed to support the variable facet *ComplexCondSpt*. So, there exist configuration dependencies between feature variations and role variations. The dependency is denoted by the relation *configDepend* between action facets and roles in our method (see figure 2). It is similar to configuration dependency on *Facet-Value* in our feature model [3], and the difference is that dependency here is between feature and role. An atomic feature-role dependency can be expressed as:

$$(action, facet{=}term) \rightarrow (role, [property{=}value]),$$

in which "*facet*=*term*" means a facet value assumption for *action*, optional "*property*=*value*" is a property value assumption for *role* (absence means configuration depending on bound of the role). Then the feature-driven configuration dependency on *ConditionSet* can be expressed as (*Search&Browse*, *ComplexCondSpt*=*true*) → (*ConditionSet*).

3.3 Design Consideration in Feature Implementation Model

Different designers can have different implementation model designs for the same feature model. However, there still some guidance. Role interaction is to provide direct guidance for program-level customization and composition. Intuitively, if different parts of a role are involved in interactions with different roles, usually they should be separated as several roles to enable program-level composition. Role granularity should also be carefully designed to maximize commonalities and localize variations. For example, the role design for the feature *BookPicShow* separates variable role *ImageFetch* from *PicShowControl* to localize the variability. If *ImageFetch* has no variability, it can be merged with *PicShowControl* for simplicity.

3.4 Role Instantiation and Role Context

From figure 2, we can see each concrete role is instantiated by a program unit. Program unit here can be a class or method in an object-oriented language. These program units correspond to roles with different types (*ope* or *res*), different interactions and different variability (mandatory, optional or variable). Operational roles are instantiated by methods, in which variability-related roles are implemented by separated method segments to be composed into the base program. Instantiation of determining role can determine the execution of other methods, so return value and the policy (specifying what return value corresponds to what decision) are also needed. Resource roles are instantiated by classes, e.g. *ImageContainer* in figure 3 can be instantiated by a Java Canvas class. As for resource roles introduced by other roles, additional initialization code is also needed to initialize it in the host class, e.g. create a Canvas object, and set its size and position, etc.

In program-level composition, necessary mechanisms of data sharing and transfer should also be established between interacting program units. In our method, role context is introduced to model inter-role interaction information. Role context is runtime information about the role, which can be accessed by other roles in interactions. For example, *ReaderCode*, *BookNo* and *Operation* are identified as the context of *BookChange*, representing code of the current reader, book number and the current operation in an execution of *BookChange*. In this example, *BorrowBook* and *ReturenBook* will inherit the context, since they are specialized roles of *BookChange*. In role instantiation, each role context should be instantiated to enable runtime context access. On the other hand, role property should also be instantiated to make the role-level customization implemented on the program-level.

The entire schema of role instantiation is presented in figure 4. We can see a role can have multiple instantiations, e.g. Role 4, and a property-value of it can determine which implementation is bound in product derivation. Other properties can be mapped to parameters of the implementation method, and then role-level customization can be embodied by program-level parameters. Role context is instantiated by constants or runtime expressions. Constant context is applicable for several roles inheriting the same abstract role. For example, the context *Operation* of *BookChange* provides description for the current operation to be recorded in the operation log, and can be instantiated as constant strings "borrow book" and "return book" in the specialized roles *BorrowBook* and *ReturenBook* respectively. Runtime expression can provide runtime context for other interacting roles, e.g. *ReaderCode* and *BookNo*. It can be any legal expressions in the runtime context of the method, e.g. "getCurrentBookNo()" (get the book number by a method call) or "bookNo" (get the book number by an object property). These expressions can be used in the glue code to share the context with interacting program units.

There are also roles with application-specific implementation, e.g. role *BorrowControl* may be different in each product. These roles can be instantiated by method or class declaration, which can be implemented by application engineers. It should be emphasized that role property and context are different mechanisms for product derivation. Role property is determined in role-level customization and affects the implementation of the same role, while role context is declared for references of other interacting roles and the value is usually determined at runtime.

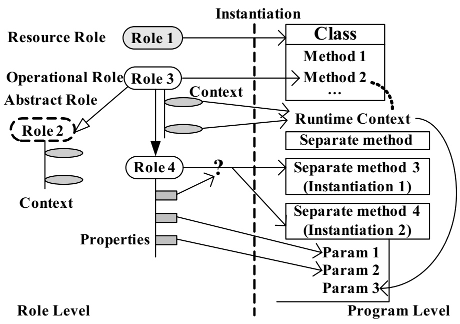


Fig. 4. Role instantiation schema

3.5 Role-Level Customization

Product derivation is first driven by application requirement, i.e. feature-level customization in SPL development. Feature customization is guided and verified by feature constraints, which is well discussed in works on feature modeling (e.g. [2][3]).

After feature customization, role-level customization can be considered for those roles residing in bound features. Feature-role dependency introduced in 3.2 will help to achieve the feature-driven role customization. Each feature-role dependency like "(*action*, *facet*=*term*) → (*role*, [*property*=*value*])" will be applied to determine the property-value of a role or bound of an optional role. If no inconsistency emerges, role-level refinement ends. Then other pending role variations, which are additional variability introduced in logical design, will be considered. For example, *FetchModel* defined in figure 3 will be determined to be *File* or *DB* completely from the design consideration (whether to use database or not).

4 Program-Level Customization and Composition

After role-level customization, variability-related program units can be selected and configured according to the role instantiation. If a role corresponds to more than one program unit, its property values are used to determine which instantiation is chosen. Program configuration is to map customized property values to program parameters, which will be transferred to the program unit by the glue code. After that, variability-related program units will be composed with each other and base programs by aspect weaving. In our implementation, AspectJ [8] is adopted to implement the program composition, so related concepts such as advice, join point are used.

Table 2. Composition rules for various role interactions

Interaction	Advice	Glue Code
Involve	**before** or **after execution**	Context acquisition, parameter preparation and transfer
Inform	**before** or **after execution**	Context acquisition, parameter preparation and transfer; Startup a new thread to execute the informed program
Determine	**around call**	Context acquisition, parameter preparation and transfer; Determine whether proceed the method or not by return value of the determining method
Access	N/A	Prepare the resource reference expression and transfer to the call to the accessing method
Introduce	resource: **Introduce** initialization: **after execution**	Embodied in the initialization code

4.1 Composition with Base Programs

Composition rules for various interaction types are listed in table 2. In each case, variability-related program unit is woven into base program. **Involve** interaction weaves the involved program by a **before** or **after execution** advice according to the interaction point. Automatically generated glue code included in the advice will acquire the context, prepare the parameters and transfer them to the involved method. **Inform** interaction is similarly treated, but a new thread will be started up to execute the informed program in an asynchronous mode. Both **Involve and Inform** interactions are implemented on the **execution** level. **Determine** interaction is implemented by **around call** advice. It is implemented on the **call** level, so that it can control other woven programs. Implementation of **Introduce** interaction includes two parts: introduce the resource object as a property of the host class; weave the initialization code into the constructor of the host class. Interaction with abstract role is cross-cutting, e.g. each role inherited from *BookChange* should inform *BookLog* and provide context information (*Reader-Code*, *BookNo* and *Operation*) for logging if it is bound. In this case, weaving and glue code will be generated for program units corresponding to all the concrete roles, each with different context information.

4.2 Composition between Variability-Related Programs

Composition between variability-related programs implements feature-dimension composition. That is program units corresponding to roles of the same feature are composed together to implement the feature, even if they are woven into different base classes. Generally, interactions between variability-related roles fall into two categories. One is the **Access** interaction between an operational role and a resource role, e.g. interaction between *PicShowControl* and *ImageContainer* in figure 3. In this case, the resource role may have another **Introduce** interaction with a base role. Programs for the two roles may be woven into different base classes, so a reference chain between them should be established to enable implementation of the **Access** interaction. It can be achieved by navigation between objects of the two classes, since the resource object can be accessed from its host by a get method added in the weaving (see table 2). So when instantiations of two roles with **Access** interaction are woven into two different classes, a navigation expression from the runtime context of the operational role will be requested from the developer, e.g. "currentBook.getAuthor()". The implementation method of accessing role can declare the resource object as a parameter. Then, in composition, the resource reference expression can be generated with the navigation expression and transferred to calls to the accessing method (see table 2).

The other category is interaction between two operational roles, e.g. interaction between *PicShowControl* and *ImageFetch* in figure 3. In this case, at most one of them may have interaction with base roles, so interaction between their instantiation can be fixed in the program. For example, invocation to *ImageFetch* can be included in the *PicShowControl* implementation, but the implementation version for *ImageFetch* may be different since it has more than one instantiation (with the same method signature) due to the role variability of *FetchMode*.

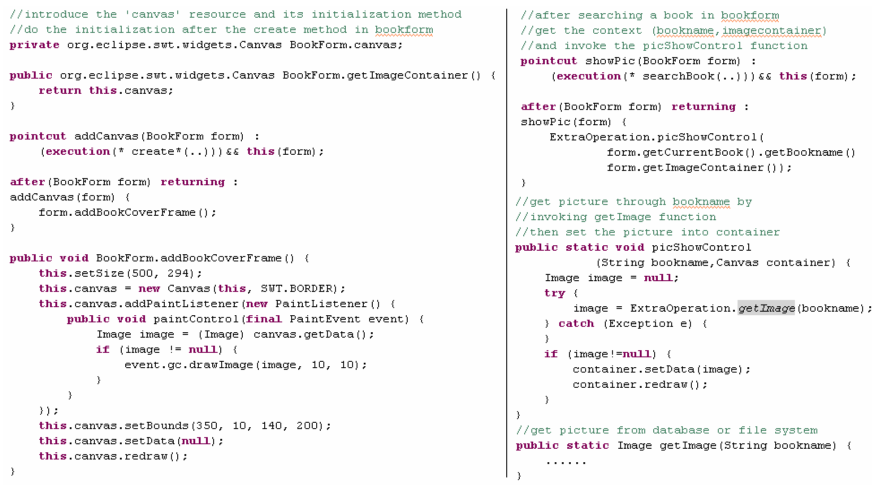
4.3 Class-Dimension Coordination

In program composition, instantiations of multiple variability-related roles may be woven into the same base method. These advices should be well coordinated to eliminate possible conflicts. In our method, two kinds of coordination are provided. One is for multiple determinations, e.g. *BorrowBook* may be determined by a new role *ReaderCheck* (check the account status) besides *BorrowControl*. In this case, multiple determinations are imposed on the same base method, so these determination rules can combined in a conjunctive mode. The other is coordination for determination and other interactions. In this case, determination is on the domination position, which can determine not only the base method, but also other interaction advices. This domination is implemented by different weaving policies: **Determine** is woven on the **call** level, while **Involve** and **Inform** are on the **execution** level.

5 Tool Support and Case Study

The method proposed in this paper has been implemented in our prototype of feature-driven product derivation tool. It is integrated with OntoFeature, the feature modeling tool developed in our previous work [3], by importing and capturing the feature list and dependencies. The tool provides editing space for each feature, supports the role customization and program-level composition by invoking the AspectJ Compiler.

Now we will demonstrate a case study of the *BookPicShow* implementation of the library management showed in figure 3. We can see the resource role *ImageContainer* has an **introduce** interaction with a base role, so it is instantiated by both an implementation class and a code segment of initialization. In this example, *ImageContainer* is instantiated by the *Canvas* class and the initialization code is to set the size and listener. Then, when *BookPicShow* is bound in the product derivation, the role *ImageContainer* can be composed by an automatically generated aspect, which declares a *Canvas* object as the inter-type of *BookForm* (the class which instantiation method of *BookSearch* resides in) and adds the initialization code after the execution of the constructor of *BookForm* (see the left part of figure 5).



```
//introduce the 'canvas' resource and its initialization method
//do the initialization after the create method in bookform
private org.eclipse.swt.widgets.Canvas BookForm.canvas;

public org.eclipse.swt.widgets.Canvas BookForm.getImageContainer() {
    return this.canvas;
}

pointcut addCanvas(BookForm form) :
    (execution(* create*(..)))&& this(form);

after(BookForm form) returning :
addCanvas(form) {
    form.addBookCoverFrame();
}

public void BookForm.addBookCoverFrame() {
    this.setSize(500, 294);
    this.canvas = new Canvas(this, SWT.BORDER);
    this.canvas.addPaintListener(new PaintListener() {
        public void paintControl(final PaintEvent event) {
            Image image = (Image) canvas.getData();
            if (image != null) {
                event.gc.drawImage(image, 10, 10);
            }
        }
    });
    this.canvas.setBounds(350, 10, 140, 200);
    this.canvas.setData(null);
    this.canvas.redraw();
}
```

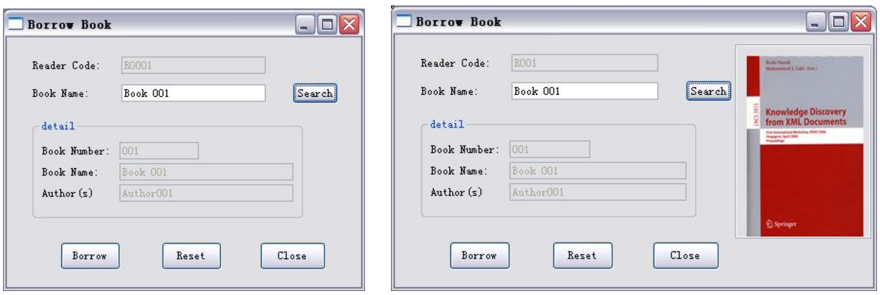
```
//after searching a book in bookform
//get the context (bookname,imagecontainer)
//and invoke the picShowControl function
pointcut showPic(BookForm form) :
    (execution(* searchBook(..)))&& this(form);

after(BookForm form) returning :
showPic(form) {
    ExtraOperation.picShowControl(
            form.getCurrentBook().getBookname()
            form.getImageContainer());
}
//get picture through bookname by
//invoking getImage function
//then set the picture into container
public static void picShowControl
        (String bookname,Canvas container) {
    Image image = null;
    try {
        image = ExtraOperation.getImage(bookname);
    } catch (Exception e) {
    }
    if (image!=null) {
        container.setData(image);
        container.redraw();
    }
}
//get picture from database or file system
public static Image getImage(String bookname) {
    ......
}
```

Fig. 5. Aspect glue code generated

On the other hand, there is an **Access** interaction between *PicShowControl* and *ImageContainer*, so a navigation expression should be given to enable the interaction. In this example, both of them are woven into the same class *BookForm*, since instantiation method of *BookSearch* also resides in it. So, the navigation expression is an empty string, because they are in the context of the same class. In our example, *PicShowControl* is composed after *searchBook* (instantiation method for *BookSearch*) by an aspect which fetches picture of searched book and set it into *ImageContainer* (see the right part of figure 5).

After feature- and role-level customization, aspects for all the bound variability-related roles can be automatically generated. Then they can be compiled together with all the base programs and variability-related instantiation programs by invoking AspectJ compiler (by *ajc* command) and a product is derived. Figure 6 shows the snapshots of the book borrow form before and after the binding of *PicShowControl*.


Borrow Book

Reader Code: R0001

Book Name: Book 001

detail

Book Number: 001

Book Name: Book 001

Author(s) Author001

Borrow Book

Reader Code: R001

Book Name: Book 001

detail

Book Number: 001

Book Name: Book 001

Author(s) Author001

Fig. 6. Book borrow form before and after the binding of *PicShowControl*

6 Related Works

In SPL researches, there has been significant effort spent on the early steps, including scope definition, domain and feature modeling and architectural design, but less attention has been paid to the implementation level [6]. Deelstra et al. [1] analyze the problems during product derivation and point out that complexity of the SPL in terms of the number of variation points and variants and implicit properties (e.g., dependencies) of variation points and variants are the two core issues. Deursen et al. [5] propose a source-level packages based method for product derivation. In the method, product is derived by packaging source-code components according to feature selections. This source code based method has no explicit model for feature interaction and feature-driven customization.

Jansen et al. [4] propose a feature based method for product derivation. Their method also introduces role model to help relate features with components, and then products can be derived by selecting a number of base components and features based on their composition algorithm. In the method, role interactions are not explicitly modeled and the composition is implemented by inheritance in object-oriented language (both base component and role are implanted by classes). Our method provides comprehensive support for role interaction modeling and implementation (by property, context, etc), and adopts a lightweight and flexible mechanism for product composition by aspect weaving.

Some researchers have noticed the potential of AOP as a SPL implementation technology. Anastasopoulos et al. [6] performed a case study to evaluate AOP as a SPL implementation technology, and drew the conclusion that AOP is especially suitable for variability across several components and whether AOP is suitable for other variability still need further study. Lee et al. [7] propose to combine feature analysis and AOP to enhance reusability, adaptability, and configurability of product line assets. They provide some good guidelines for AOP based SPL assets development by considering commonality and variability, dependency and binding time. However, the method lacks an intermediate level to clarify the connection between feature and program implementation, so can not support feature-driven program customization and composition.

7 Conclusion and Discussion

In this paper, we propose a product derivation method in which feature-driven program-level customization and composition are supported by feature implementation modeling, instantiation and aspect weaving in AOP. The main contribution of this paper is enabling feature-driven program-level customization and composition for product derivation by introducing an intermediate feature implementation model between feature model and program implementation along with corresponding customization and instantiation. However, our method doses not cover the issue of feature-driven DSSA (Domain Specific Software Architecture) design. In fact, domain-level design and implementation are assumed to have been done. Our method provides a mechanism of feature implementation design and instantiation to map feature-level customization to program-level configuration and composition. It is an implementation technology for product derivation in SPL. In the future research, we will focus on more systematic and comprehensive support for feature-driven implementation design and SPL evolution management.

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. 2006AA01Z189 and 2007AA01Z125.

References

1. Deelstra, S., Sinnema, M., Bosch, J.: Experiences in Software Product Families: Problems and Issues During Product Derivation. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154. Springer, Heidelberg (2004)
2. Zhang, W., Mei, H., Zhao, H.: Feature-driven requirement dependency analysis and high-level software design. Requirements Eng. 11, 205–220 (2006)
3. Peng, X., Zhao, W., Xue, Y., Wu, Y.: Ontology-Based Feature Modeling and Application-Oriented Tailoring. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039. Springer, Heidelberg (2006)
4. Jansen, A.G.J., Smedinga, R., van Gurp, J., Bosch, J.: First class feature abstractions for product derivation. IEE Proc.-Softw. 151(4) (2004)
5. van Deursen, A., de Jonge, M., Kuipers, T.: Feature-Based Product Line Instantiation Using Source-Level Packages. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379. Springer, Heidelberg (2002)
6. Anastasopoulos, M., Muthig, D.: An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: Bosch, J., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107. Springer, Heidelberg (2004)
7. Lee, K., Kang, K.C., Kim, M., Park, S.: Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In: SPLC 2006. IEEE Computer Society, Los Alamitos (2006)
8. AspectJ Team. AspectJ Project, `http://www.eclipse.org/aspectj/`