

# Towards Feature-oriented Variability Reconfiguration in Dynamic Software Product Lines

Liwei Shen, Xin Peng, Jindu Liu and Wenyun Zhao

School of Computer Science, Fudan University, Shanghai, China  
{shenliwei, pengxin, 09212010014, wyzhao}@fudan.edu.cn

**Abstract.** Dynamic Software Product Line (DSPL) provides a new paradigm for developing self-adaptive systems with the principles of software product line engineering. DSPL emphasizes variability analysis and design at development time and variability binding and reconfiguration at runtime, thus requires some kinds of variability mechanisms to map high-level variations (usually represented by features) to low-level implementation and support runtime reconfiguration. Existing work on DSPL usually assumes that variation features can be directly mapped to coarse-grained elements like services, components or plug-ins, making the methods hard to be applied for traditional software systems. In this paper, we propose a feature-oriented method to support runtime variability reconfiguration in DSPLs. The method introduces the concept of role model, an intermediate level between feature variations and implementations to improve their traceability. On the other hand, the method involves a reference implementation framework based on dynamic aspect mechanisms to implement the runtime reconfiguration. We illustrate the process of applying the proposed method with a concrete case study, which helps to validate the effectiveness of our method.

**Keywords:** Dynamic software product line, self-adaptation, dynamic AOP, variability binding, reconfiguration

## 1 Introduction

Software-intensive systems in areas like pervasive computing and online service systems are required to be more adaptive nowadays. Rather than behaving constantly, these systems, also called self-adaptive systems, can automatically adapt their behaviors at runtime based on the environment and guided by objectives and needs of stakeholders [1].

In the software reuse community, Dynamic Software Product Line (DSPL) [2] has been proposed as an effective paradigm to develop self-adaptive systems with the principle of software product line (SPL) engineering. DSPL identifies the reusable and dynamically reconfigurable core assets at development time which are explicitly modeled as dynamic variability. At runtime, DSPL application proposes to configure and reconfigure runtime instances by the variability customization, which means to adapt the binding decisions of the variations within the current system during execution. The business logic of a DSPL application covers the adaptable behaviors

which can be represented as a domain model, usually as a feature model [10, 11]. As a result, the dynamic reconfiguration strategies can be obtained and specified in a higher feature level rather than the lower program level, which makes it easily be validated and understood by the system users.

In order to implement the feature-oriented variability reconfiguration, DSPL requires some kinds of variability mechanisms to map high-level feature variations to low-level implementations. Existing work usually assumes that variation features can be directly mapped to coarse-grained implementation elements like services [7, 8], components [4] or plug-ins [3]. However, the variability traceability from features to implementations in the traditional software systems may be more complicated. For example, a single feature can be implemented by multiple program units, while a program unit may also contain the functions for several features. Existing feature-based methods do not provide the traceability naturally due to the big gap between the problem space and the solution space [12, 13]. Thus, the program-level reconfiguration driven by the feature level variation binding is hard to be performed.

In this paper, we propose a feature-oriented method to support runtime variability reconfiguration in DSPLs. The method introduces the concept of role model, an intermediate level between feature variations and implementations to improve their traceability. In a role model, each feature is implemented by a set of roles as well as various role interactions, and roles will be further instantiated by elements in the implementation level. In particular, a special type of roles, called control roles, is introduced to manage the dynamic reconfiguration upon feature variations. On the other hand, our method involves a reference implementation framework based on dynamic aspect mechanisms to implement the runtime reconfiguration. Currently, we adopt *JBoss-AOP* [18], a popular mechanism supporting dynamic aspect weaving. Following our method, we conducted a concrete case study on a course selection system, and the preliminary results help to validate the effectiveness of our method.

The remainder of this paper is organized as follows. Section 2 gives a background introduction to the DSPL and analyzes research problems in feature-oriented variability reconfiguration. Section 3 describes the role model as well as its capability to support the runtime reconfiguration. Then Section 4 introduces our reference implementation framework based on dynamic aspect mechanisms. Section 5 presents the case study with the discussion, and Section 6 introduces related work. Finally Section 7 draws the conclusion and plans our future work.

## **2 Background and Problems for Variability Reconfiguration**

In this section, we will first briefly introduce the background of a DSPL. Then the working example of a course selection system will be given out to derive the research problems when realizing the feature variability reconfiguration.

### **2.1 Background of DSPLs**

Complying with a traditional SPL, the feature model of a DSPL provides an integrated business view emphasizing on possible variations and changes in its runtime behaviors. Thus, the variability in DSPLs is bound or unbound during

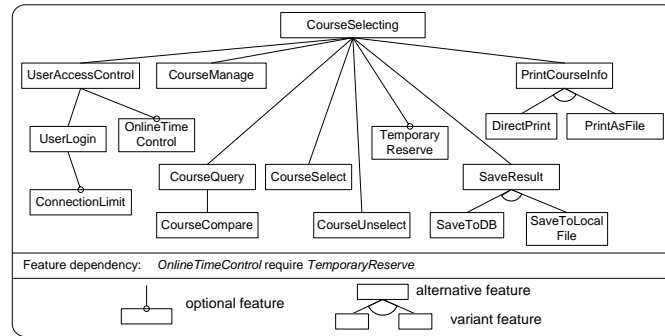
runtime and the binding decisions on the variations may change several times in its lifetime [2].

In our method, we consider two kinds of variation points (i.e. adaptation points at runtime) in feature models which are optional and alternative. Since the binding status of the variations can be changed several times during execution, all of their corresponding implementations should be incorporated into the application initially. At runtime, whether they are available will depend on the reconfiguration strategies generated from the adaptation rules in a specific context.

We think an important engineering principle that DSPL can bring to self-adaptive systems is the feature dependency and constraint mechanism. By feature constraints, we can represent *require* or *exclude* dependencies among feature variations, and combine them into runtime adaptation controls to help to ensure consistent and complete feature bindings and reconfigurations.

## 2.2 DSPL Example of a Course Selection System (CSS)

The Course Selection System is a DSPL example whose feature model (in Figure 1) as well as its adaptation rules (in Table 1) is identified before the system is running. The system is endowed with the capability of self-adaptation so that it can provide a stable online service facing the course-selecting demand from thousands of students in a campus. The adaptation capabilities are formalized as the ECA (*On Event If Condition Do Action*) [16] rules which indicate the operations upon the dynamic variation points in the feature model.



**Fig. 1.** Feature model of the course-selecting system

The self-management capability helps to generate the variations in the business model which represents the possible configurations that the system may behave at runtime. For example in the figure, the feature *OnlineTimeControl* is an optional feature that can be bound or unbound according to the changing concurrent accessing number specified in the first rule. *TemporaryReserve* which can keep the unsaved user operations for a period of time if the user is disconnected is required by *OnlineTimeControl*. It means the former cannot be bound if the latter is not bound. *ConnectionLimit* is another optional feature whose binding status depends on the available server memory. *SaveResult* is an alternative feature whose variants are *SaveToDB* and *SaveToLocalFile* separately. Thus, the saving mode can be changed at runtime conforming to the availability status of the database which is evaluated through the response time shown in the third rule. *PrintCourseInfo* is similar with the

previous alternative feature that its binding strategy of its variants depends on the availability of the printer (the forth rule).

**Table 1.** The ECA rules for CCS

ON	IF	DO
the concurrent accessing number (CAN) changes	CAN > 500	bind <i>OnlineTimeControl</i>
	CAN < 450	unbind <i>OnlineTimeControl</i>
the memory utilization (MU) changes	MU > 90%	bind <i>ConnectionLimit</i>
	MU < 80%	unbind <i>ConnectionLimit</i>
database response time (DRT) changes	DRT > 3s	bind <i>SaveToLocalFile</i>
	DRT < 1s	bind <i>SaveToDB</i>
printer state (PS) changes	PS = out-of-service	bind <i>PrintAsFile</i>
	PS = in-service	bind <i>DirectPrint</i>

### 2.3 Research Problems in Feature-Oriented Variability Reconfiguration

During runtime, the DSPL application should monitor the current situation and activate the corresponding feature variability reconfiguration specified by the ECA rules. However, when transferring the reconfiguration from the feature level to the implementation level, the following problems may emerge.

On the one hand, the complex mapping between the features and the implementation artefacts poses difficulty for the dynamic reconfiguration. Under the circumstance, a feature is not directly mapped to the coarse-grained component, service or plug-in. However, a single feature variation may influence several fine-grained program units. For example, if the feature *OnlineTimeControl* is bound to be available, it will first introduce a list which is used to store the remaining online time of each access to the module implementing *UserAccessControl*. Furthermore, it will also append an operation to start the timing as soon as a new access is permitted. In addition, the feature will activate a thread which checks the time list all the time to find out and stop the overtime access. Thus, without a clear traceability between the two levels, what to do to reconfigure in the program artefacts is hard to be specified.

On the other hand, the underlying runtime collaboration between the features may also influence the execution of the dynamic reconfiguration. For instance, if *ConnectionLimit* is bound, it can modify the behavior of *Userlogin* by determining whether to execute it or not although the collaboration is not explicitly illustrated in the feature model. If it is not bound, the constraint will not take effect. Another example is that *UserAccessControl* may invoke *OnlineTimeControl* to initialize the timing for an access if the latter feature is bound. The collaborations especially between the common features and the variable features vary in different scenarios and can be dynamically established or revoked during runtime. Therefore, without a comprehensive description of the various feature collaboration, the reconfiguration is hard to be applied in the program level, i.e. what kind of collaborations and how they can be reconfigured.

Existing coarse-grained mapping methods are not suitable for capturing the decomposition of the feature's responsibility as well as the underlying collaborations. Therefore, an effective mapping method which provides explicit traceability from the business model to the implementation artefacts is desired. In addition, a mechanism towards the program level adaptation driven by the dynamic feature binding is consequently necessary.

Based on the considerations, we will introduce the role model as well as the reference implementation framework in Section 3 and 4 separately.

### 3 Role Model of DSPLs

In this section, we will first introduce the basic concepts of the role model by illustrating its meta-model. The included role interactions and their semantics are identified. Then we will present the role-level interaction reconfiguration which is supported by the special control roles.

#### 3.1 The Role Meta-Model

Role model is the logic design model for the implementation of features [18]. It is regarded as a type of domain architecture which relates the business features to the actual program artefacts. Figure 2 depicts the role meta-model together with the features and the code implementations.

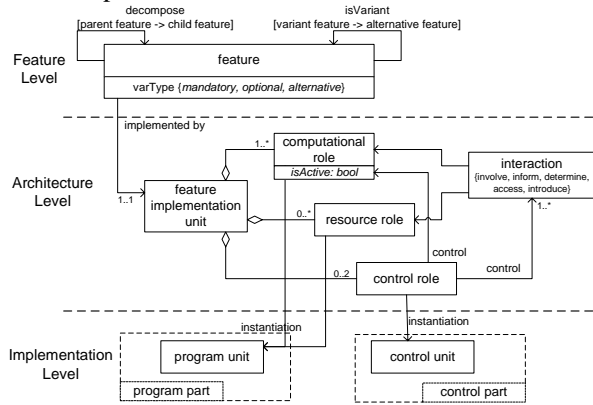


Fig. 2. Meta-model of the role model

In the feature model, feature is the basic element which represents the business operation of a DSPL. Mandatory features, optional features and alternative features are expressed separately. Usually, parent features can be decomposed into the child features which are organized in a tree-like style. Furthermore, variants are specific features that are related to the parent alternative feature with the relationship *isVariant*.

In the architecture level, a feature is implemented by a feature implementation unit which is combined with different kinds of roles. The concept of role here is similar to the *responsibility* in [12] and the *role* in [9]. Compared with features, roles reserve the knowledge of features as well as endow features with the implementation aspects, so it can support the mapping from features to program artefacts in a more natural way [19].

From the meta-model, there are three kinds of roles with separate design concerns. Firstly, a computational role denotes a logical unit or a responsibility which should be taken by the program units for feature implementation. It refines the function of a feature into a finer-grained level so that the existing feature tangling and scattering problems can be alleviated. Furthermore, a computational role has a property *isActive*

which claims that the role runs in a recurrent way and requests other's services if the value is true, otherwise to be invoked by others if the value is false. An active role is common in an information system that a lot of specific events are triggered by the active inspection. Secondly, a resource role represents the specific internal or external entity necessary for the implementation of a feature. It is usually cooperated with other computational roles to fulfill a certain business goal. For example, it can be a widget in the user interface and then a computational role can specify its content, or a data structure used to store information which can be read or written by other computational roles. Thirdly, a control role is of much difference with the previous two roles. It takes the responsibility in two respects. On one hand, it helps to manage the status of the interactions (they will be described later) between the previous two roles. Thus, it can control whether a specific interaction can be established or revoked during runtime. On the other hand, it can also influence the active role to determine its running status, i.e. to start it or to suspend it.

A feature may be implemented by a set of identified roles which organize a feature implementation unit. Usually, a feature is certainly to be refined as one or more computational roles. It also involves some resource roles on condition that the feature works on some entities (UI widget, data structure, etc). The control role is not an indispensable element. It exists only if the feature implementation unit corresponds to a feature with variability. This kind of roles takes effect according to the binding strategy of the features.

Furthermore, we do not define the variability property to the roles in the meta-model since all the roles should be included in the product but available at different times. However, for the sake of clearness, we distinguish the roles as base roles and variability-related roles. The former resides in the feature implementation unit which relates to a mandatory feature, while the latter is refined from the variation features.

In the implementation level, the computational roles as well as the resource roles can be instantiated by program artefacts which compose the program part of a DSPL application. The artefacts may be components, classes, methods, code segments or configuration files with various granularities. However, they should be organized as an encapsulated entity especially for the fine-grained units. Sometimes several roles will be instantiated with the artefacts contained in a same encapsulation which is the result of feature tangling. On the other hand, the control role is instantiated by the control unit that contains the operations for managing the role interactions and the active computational roles. It resides in the control part within the DSPL implementations.

Furthermore, there exist various kinds of interactions between the roles. They indirectly denote the underlying collaborations between the features in order to reach the system business goals. Following our previous work in [18], we list the five identified interactions as well as their descriptions in Table 2. The *involve* and *inform* interactions are the functional dependencies between two computational roles, however in different invoking mode. The *determine* interaction is special that it is used to manage the normal execution of the target role by means of the semantic of the source role. The *access* and *introduce* interactions are built between a computational role and a resource role with opposite directions.

It should also be noticed that the *control* links from a control role are not regarded as a kind of role interaction as listed. In fact, it is of great importance for the role-level

interaction reconfiguration based on the feature-level variability reconfiguration which will be represented in the next subsection.

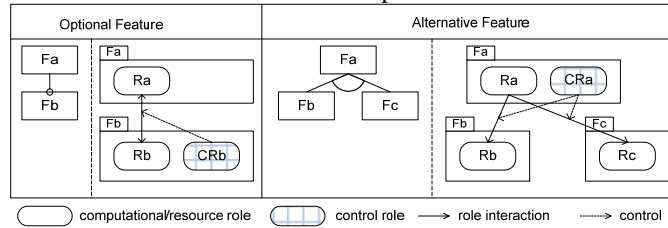
**Table 2.** Descriptions of the role interactions [18]

role interaction	description
involve	a computational role requires a function from another computational role in a synchronous mode
inform	a computational role activates the executing of another computational role in an asynchronous mode
determine	a computational role decides the behavior of another computational role by allowing or rejecting the operation of the target role
access	a computational role accesses a resource role by reading, writing or modifying it
introduce	a resource role is introduced into an implementation unit of another computational role to be a sub-element

### 3.2 Role-Level Interaction Reconfiguration

During the adaptation process, the feature variation reconfiguration is further mapped to the role model. Under this circumstance, the variability-related roles can be adapted to be available or unavailable when the runtime reconfiguration is performed. Since the roles cannot be removed from the running application, the management of the role interactions supports the reconfiguration, i.e. the interactions are variable and the dynamic establishing or revoking of the role interactions can change the application's behavior during runtime. For example, if a base role *A* has an involve interaction towards an optional-related role *B*, the interaction should be built when the optional target does not exist anymore.

In our method, the interaction reconfiguration is supported by the control roles. Figure 3 depicts the role-level interaction reconfiguration patterns by the control roles which are related to the two kinds of variation points in the feature model.



**Fig.3.** Role-level reconfiguration patterns by control roles

For the sake of simplicity, we assume that each feature is implemented by one computational or resource role. Furthermore, in the corresponding feature implementation unit, there exists a control role (*CRx*) if the feature is of variability. The semantics of the patterns are as follows.

The pattern for the optional feature: Suppose *Fb* is an optional feature whose parent feature is *Fa*. Under the separate situations of the interaction scenarios and the role types, there may exist all the five types of interactions between *Ra* and *Rb*, e.g. usually *Ra* involve/inform/access *Rb*, or *Rb* determine/introduce *Ra*. The control role

*CRb* manages the existence of the interaction during execution, i.e. establish it when feature *Fb* is bound, or revoke it when *Fb* is unbound.

The pattern for the alternative feature: Suppose *Fa* is an alternative feature and its variants are *Fb* and *Fc*. Since the invocation to *Fa* may be transferred to any of its variant, *Ra* can have involve or inform interactions towards *Rb* and *Rc*. However, the underlying constraint indicates that the two interactions are mutually exclusive, which means only one interaction can be established at a specific time. Therefore, the switch between the interactions is controlled by *CRA*. Its operation includes two respects of actions, i.e. remove the interaction towards one role and establish the interaction towards another role.

Furthermore, the feature dependency should also be considered. It has nothing to do with the reconfiguration patterns but to put the dependent reconfigurations together in order to reach a valid runtime instance. For example, if an optional feature requires another optional feature, the dependency will indicate that the reconfigurations of the different feature implementation units should be performed at the same time.

Besides, the control role also takes the responsibility to start or suspend an active role in a variability-related feature implementation unit. For example, an active role *Ra* is refined from an optional feature *Fa*, and *Ra* continuously checks the modification of an information list to trigger the events to be handled. Supposing *Fa* is not bound, *Ra*'s running is meaningless and also wastes the system resource. Thus, it should be stopped or suspended. On the contrary, if *Fa* is to be bound, *Ra* should be started as soon as possible. Thus, it is the control role that manages the running status of an active role.

## 4 The Reference Implementation Framework of DSPLs

The feature-oriented variability reconfigurations should be realized in the implementation layer with the support of the role model. In this section, we will describe the referent implementation framework which adopts *JBoss-AOP* as the dynamic aspect mechanism.

### 4.1 Dynamic-AOP in *JBoss-AOP*

Dynamic AOP is a powerful technique to realize the dynamic program adaptation [14, 15]. Similar with the traditional static AOP, it also involves the concepts of the binding from advice to pointcut. However, the key characteristic is that the binding can be decided during runtime, i.e. the inclusion of the binding can be changed.

In our method, we adopt *JBoss-AOP* [15] as the underlying mechanism for the dynamic aspect weaving. In *JBoss-AOP*, an aspect is defined by an interceptor and a pointcut. The interceptor is the same as the advice and it is programmed as a *Java* class. The aspect bindings are prepared to be included.

When the dynamic aspect is to be woven, *JBoss-AOP* will dynamically insert or remove the aspect binding through *AspectManager*. The binding establishing process can be described in the following steps: 1) create a new aspect binding by a given name; 2) declare the pointcut of the base program in the created binding; 3) declare the interceptor to the binding. On the other hand, the existing named aspect can also be removed by means of the *AspectManager*.

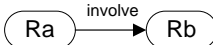
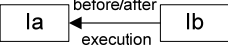
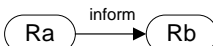
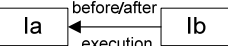
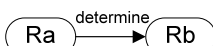
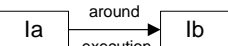
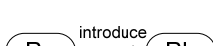
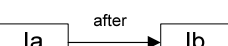
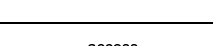
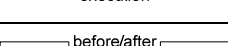


## 4.2 Implementation-Level Composition Patterns

The roles are instantiated by the program units or control units. As mentioned in Section 3, the program units can be any artefact in different granularity, including the encapsulated components, classes, or scattered code segments. In our method, we claim the following development principles for the implementations in order to support the runtime reconfiguration. Firstly, the base roles and the variability-related roles should be instantiated by different program units. In particular, different base roles can reside in the same unit while different variability-roles are suggested to be located in separate entities. Secondly, the base roles are requested to correspond to the program artefacts whose granularity is bigger than method so that they can be woven by aspects, i.e. the pointcut can be defined around the methods declared. Thirdly, the program which instantiates a variability-related role should be encapsulated as an interceptor which is a regular *Java* class and implements the *Interceptor* interface.

When preparing a dynamic aspect, the expression of the pointcut as well as the content of the interceptor depends on the type of the role interactions. Table 3 illustrates the composition patterns for the program artefacts separated by the various interaction types. In this mapping, we simply assume that each role is instantiated by a corresponding code encapsulation.

**Table 3.** Composition patterns for different role interactions

role interaction	composition pattern	<i>JBoss-AOP</i> implementation
		pointcut: before or after the execution of <i>Ia</i> interceptor: <i>Ib</i> , runs in a synchronous mode (the same thread)
		pointcut: before or after the execution of <i>Ia</i> interceptor: <i>Ib</i> , runs in an asynchronous mode (a new thread)
		pointcut: around the execution of <i>Ib</i> interceptor: <i>Ia</i> , decides whether to proceed <i>Ib</i> 's execution or not
		pointcut: after the execution of <i>Ib</i> interceptor: <i>Ia</i> , added into <i>Ib</i> as a new member, be declared and instantiated after the execution (constructor) of <i>Ib</i>
		pointcut: before or after the execution of <i>Ia</i> interceptor: <i>Ib</i> , the direct accessing of the resources

The direction of the aspect weaving (from the interceptor to the pointcut) does not always conform to the direction of the original interaction. For example in the first two rows, if *Ra* has an involve or inform interaction towards *Rb*, *Ia* is woven which means *Ib* is able to adapt the behavior of *Ia* by appending additional functions in an obliviousness way. However, the woven logic can be performed in different modes. Usually, if a role is involved, the base program has to be suspended until the finishing of the interceptor's execution. On the contrary, the base program notifies the interceptor to run in a new thread and continues its execution if a role is informed. In particular, the introduce interaction adopts a simple weaving mode since there is no *intertype* support in *JBoss-AOP* like that in *AspectJ* [17]. Thus, the resource is introduced to be a member of the target unit by appending the code segment which is used to declare and instantiate the resource after the execution of the target implementation, usually after the constructor.

During runtime, the interactions between the roles can be established or revoked. Therefore, the binding statuses of the dynamic aspects towards the various interactions are decided by the implementations of the control roles.

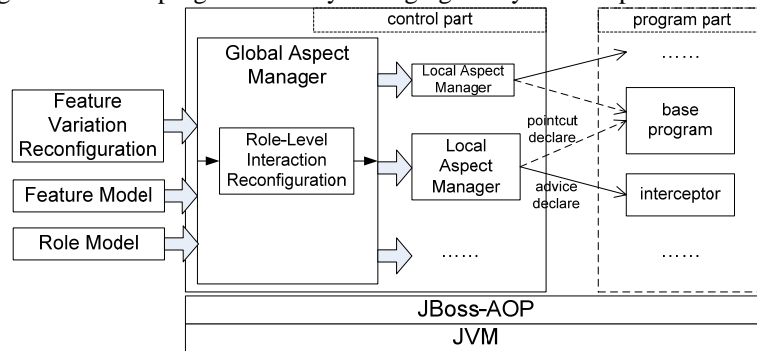
### 4.3 The Reference Implementation Framework

The reference implementation framework based on *JBoss-AOP* is depicted in Figure 4. It is noticeable that the framework only works for the reconfiguration realization. Therefore, how as well as when the adaptation needs are generated is out of scope.

When encountering a situation that triggers reconfiguration, the strategy of the feature variability reconfiguration is obtained based on the predefined ECA rules. The strategy is the input to the framework and is handled by the two special ingredients which are the global aspect manager and the local aspect manager.

Each local aspect manager is the instantiation of the control role in a single feature implementation unit. It is responsible for establishing or removing the dynamic aspect binding through the steps mentioned in Section 4.1. The local aspect manager is an encapsulated entity composed of a set of operations. The semantics of the operations follow the reconfiguration patterns in Figure 3. For example, in the case of an optional feature, a local aspect manager has one operation to bind the dynamic aspect and the other operation to remove it. As for an alternative feature, each operation has to first remove the existing aspect binding and then to weave the one related to the selected variant in a sequence. In addition, the local aspect managers should be implemented before the system is running. Which operation will be invoked depends on the reconfiguration strategy which is managed by the global aspect manager.

The global aspect manager, on the other hand, is an independent entity that is responsible for distributing the reconfiguration tasks. As soon as it receives the strategy of the feature variability reconfiguration, the global aspect manager validates the strategy based on the feature model as well as the feature dependencies defined. Then the strategy is transferred to derive a set of role-level interaction reconfigurations. The local aspect managers in the corresponding feature implementation units can be identified based on the feature traceability knowledge. Finally it automatically invokes the corresponding operations to implement the reconfiguration at the program level by managing the dynamic aspects.



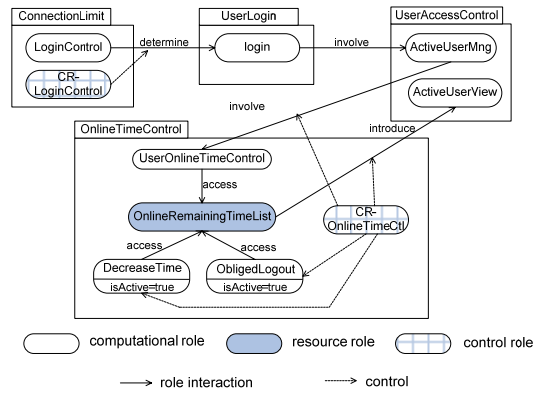
**Fig.4.** The reference implementation framework based on *JBoss-AOP*

## 5 Case Study

In this section, we will go back to the course selection system in Figure 1 to give out the role model for the DSPL application. Based on it, the program level artefacts following the reference implementation framework are also illustrated.

### 5.1 The Role Model for the Course Selection System (CCS)

Based on the role meta-model introduced in Section 3, we have constructed the role model for CCS, whose segments are depicted in Figure 5.



**Fig.5.** Segments of the role model for CSS

Figure 5 represents the role model corresponding to the *UserAccessControl* subtree in Figure 1. In the feature implementation unit of *ConnectLimit*, we have identified a control role *CR-LoginControl*. It aims to control the determine interaction towards the role *login* in another implementation unit. Furthermore, in the feature implementation unit of *OnlineTimeControl*, the feature is implemented into several roles. Supposing the corresponding feature is bound at runtime, a resource role *OnlineRemainingTimeList* which represents an additional list column is introduced to *ActiveUserView*. The interaction indicates to add a column to the list in the form where the system administrator can check the left time for each user connection. In addition, *UserOnlineTimeControl* is involved by *ActiveUserMng* once there is a new connection. When it is invoked, it will access *OnlineRemainingTimeList* to insert a record of the new access along with its maximal remaining time such as 15 minutes. On the other hand, *ObligatedLogout* is an active role which will continuously access *OnlineTime* to discover the connection that has expired. *DecreaseTime* is also an active role whose responsibility is to modify the time value in the list every three minutes. In the feature implementation unit, the two interactions as well as the two active roles are managed by *CR-OnlineTimeCtl*.

### 5.2 Reconfiguration based on the Reference Implementation Framework

Before the DSPL application is running, the artefacts for the dynamic reconfiguration should be implemented. As discussed in Section 4.2, the variability-related roles

should be instantiated by the interceptors. We take the resource role *OnlineRemainingTimeList* as an example. Figure 6 depicts the code snippet of the interceptor which instantiates the role.

```
public class TimeListInterceptor implements Interceptor {
    @Override
    public String getName() {
        return "TimeListInterceptor";
    }
    @Override
    public Object invoke(Invocation invocation) throws Throwable {
        Object result = invocation.invokeNext();
        //get the target object usersList
        Object object = invocation.getTargetObject();
        if (object instanceof UsersList) {
            UsersList usersList = (UsersList) object;
            //introduce the limit time column to the show table
            TableColumn loginDateColumn = new TableColumn(usersList.userlistTable, SWT.LEFT);
            loginDateColumn.setText("LimitTime");
            loginDateColumn.setWidth(130);
            //initialize the OnlineTimeList
            AspectPanel.OnlineTimeList.clear();
            Object[] userNames = usersList.getUsers().keySet().toArray();
            for (Object userName : userNames) {
                AspectPanel.OnlineTimeList.put(userName.toString(), AspectPanel.defaultTime);
            }
        }
        return result;
    }
}
```

**Fig.6.** Code snippets of the role *OnlineRemainingTimeList* (as an interceptor)

The resource role is introduced into a user-visible form which is developed by SWT (the Standard Widget Toolkit for *Java*). The interceptor then includes a method that appends the codes about the declaration and initialization of the widget after the execution of the pointcut.

On the other hand, *CR-OnlineTimeCtl* should be instantiated as a local aspect manager which contains the operations to establish or revoke the related aspect bindings. Figure 7 depicts the code snippet for the control role.

```
public class CR_OnlineTimeControl {
    AspectManager manager = AspectManager.instance();
    //This function wave two Interceptor.
    public void bindOnlineTime() throws ParseException {
        //1.bind the OnlineTimeInterceptor
        //Here the pointcut is instrument using JBoss AOP expression "execution(...)"
        AdviceBinding timeBinding = new AdviceBinding("OnlineTimeInterceptor",
            "execution(public void swt.UsersList->registerUser(*))", null);
        timeBinding.addInterceptor(OnlineTimeInterceptor.class);
        manager.addBinding(timeBinding);
        //2.bind the "TimeListInterceptor"
        AdviceBinding timeListBinding = new AdviceBinding("TimeListInterceptor",
            "execution(public * swt.UsersList->userToArray(*))", null);
        timeListBinding.addInterceptor(timeListInterceptor.class);
        manager.addBinding(timeListBinding);
        //3.begin to force users who are timeout to logout
        startObligedLogout();
        //4.start to count down users' online time
        startDecreaseTime();
    }
    //This function remove the bindings and stop timer
    public void unbindOnlineTime() throws ParseException {
        //1.stop forcing users out
        stopObligedLogout();
        //2.stop timer
        stopDecreaseTime();
        //3.remove the binding of OnlineTimeInterceptor
        if (manager.getBindings().containsKey("OnlineTimeInterceptor"))
            manager.removeBinding("OnlineTimeInterceptor");
        //4.remove the binding of timeListInterceptor
        if (manager.getBindings().containsKey("timeListInterceptor"))
            manager.removeBinding("timeListInterceptor");
    }
    // startDecreaseTime(), stopDecreaseTime(),startObligedLogout() and stopObligedLogout() are eliminated here
}
```

**Fig.7.** Code snippets of the control role (*CR\_OnlineControl*)

The local aspect manager is instantiated as a Java class. It first references to the single instance of *AspectManager* provided by the *JBoss-AOP* API (it is not the same as the global aspect manager in the framework). The two operations are represented by the two methods, i.e. *bindOnlineTime* and *unbindOnlineTime*. In the former method, two new aspect bindings are created which can be seen in the first two steps. In sequence, the last two steps offer the instructions to start the programs which instantiate the active roles. On the other hand, the latter method is responsible for removing the named bindings and stopping the active roles' execution.

During execution, we assume the situation that the concurrent accessing number to CCS has exceeded 500. Therefore, the adaptation is triggered and the action of the first ECA rule in Table 1 (bind *OnlineTimeControl*) is to be performed. On the same time, due to the feature dependency, the optional feature *TemporaryReserve* should also be bound.

The feature variability reconfiguration strategy is input into the framework and the reconfiguration is handled by the global aspect manager. During the process, it firstly identifies the local aspect managers for *OnlineTimeControl* and *TemporaryReserve*. Then it invokes the corresponding methods to accomplish the target of the runtime reconfiguration, e.g. invokes *bindOnlineTime* method for realizing the binding of *OnlineTimeControl*. As a result, the behavior of the DSPL has been modified without recompiling the application.

### 5.3 Discussion

Based on the preliminary results from the case study, the proposed method can be regarded as an effective way to help the feature-oriented variability reconfiguration in DSPLs.

First, the role model helps to establish the dynamically reconfigurable artefacts based on the refinement of the features. At the same time, it is also used to clarify the complex mapping between the features and the implementation programs. When refining, we use the computational and resource roles to cover the responsibility of a feature. After that, the roles are instantiated by program artefacts. Thus, at the application construction time, the role model can indicate the developers about the artefacts that should be prepared for the dynamic reconfiguration, e.g. implementation for the role *OnlineRemainingTimeList* as an interceptor and the implementation for the control role *CR-OnlineTimeCtl*. In addition, during the reconfiguration process, the artefacts can be located based on the clear traceability so that the reconfiguration in the implementation level can be correctly implemented conforming to the feature level strategy.

Second, the role interactions in the model contribute to clarify the underlying runtime collaboration between the features. Besides the identified feature dependencies, the semantic relationships between the features should also be captured. Therefore, the role interactions describe the behavior of the application and explain how the features can be collaborated to achieve the business goal in a finer-grained view. On the other hand, we conclude the patterns for the different role interaction types based on the AOP technique (see in Table 3) although the binding directions as well as the definitions of the aspect elements vary between each other. Thus, the reconfiguration on the feature variations can be transferred into the role level as the

decision on the binding status of a set of unified dynamic aspects, further identifying the reconfiguration scope in the program level. Furthermore, the special control roles are necessary in the DSPL applications that they help to interpret the reconfiguration from the features to the roles.

In a word, the benefits provided by the role model help us to draw a systematic process for the realization of the runtime adaptation, i.e. from the feature-level variation reconfiguration, to the role-level interaction reconfiguration, and finally to the implementation-level code adaptation.

We also propose a reference implementation framework to support the reconfiguration realization. The framework claims the program patterns for the role interactions including the interceptors and the local aspect managers which should be developed before the system is running. Based on it, the program-level reconfiguration can be performed in a manageable way. On the other hand, the dynamic aspect mechanism of *JBoss-AOP* is able to change the application's behavior without intervening the system running.

However, the method introduced still has several limitations. First, it may have difficulty for managing the artefacts when encountering large scale applications. Under the circumstances, the domain model may contain thousands of features, which will make the number of the roles as well as the program artefacts expand to a degree hard to handle. Second, the role model is designed in an ad-hoc way. Furthermore, the codes for the dynamic aspect binding are also developed by experienced developers with the knowledge of *JBoss-AOP*. So, it lacks a supporting tool which can ease the job for practical use. Third, the example DSPL application in our paper is simple that it cannot cover all the applied capability for adaptation. In fact, we have not considered the scenarios where conflicts will emerge in the reconfiguration process.

## 6 Related Works

The method in this paper is the extension of our previous work in [18]. In that work, the role model is introduced to ease the product derivation in the product line with static variability. The role interactions are classified and implemented using *AspectJ*. However, our method applies the role model for a single DSPL application where there is also the problem of the complex mapping between the features and the program implementations. Under the circumstance, some new characteristics have been addressed. First, we introduce a new kind of control role which is of great importance to the role-level reconfiguration. Second, we adopt the dynamic-AOP technique (*JBoss-AOP*) to support the implementation-level adaptation since *AspectJ* lacks the capability for the dynamic aspect weaving. Third, we propose a reference implementation framework for the realization of the runtime reconfiguration.

Our work is close to the dynamic product reconfiguration. Lee et al. [5] propose a systematic approach to developing dynamically reconfigurable core assets in product line engineering as well as a reconfigurator model to manage the product configuration at runtime. In their method, the feature binding analysis takes an important role in identifying the granularity of the configuration units and the corresponding binding time. On the other hand, the reconfigurator including the master and local configurators is introduced to perform the consistent feature

variation binding during runtime. In addition, Lee et al. [6] further provide a formal representation mechanism to analyze and specify the features that may vary as a part of reconfigurations within a family of products, thus supporting the consistent reconfiguration. Our method also aims at the goal of the dynamic product reconfiguration driven by the feature variation binding. However, it differs in the following aspects: We focus on the reconfiguration of a single DSPL application and all the feature variations are incorporated into the product initially. Furthermore, we also involve the activity to analyze features but in an opposite direction that we refine them to imply the program artefacts in the finer granularity level based on the role model.

There are other works concentrating on the dynamic reconfiguration upon the different kinds of program artefacts driven by the feature variation bindings. Trinidad et al. [4] propose a method to map the feature models onto the component architecture for building a DSPL. The mapping is direct and the self-adaptation can be realized by the dynamic connection between the specific components. Wolfinger et al. [3] propose their method to support adaptation by means of the combination of the product line engineering and the plug-in techniques. The adaptation is then realized by loading and unloading the plug-ins at runtime. Lee and Kotonya [7, 8] introduce their work on the service-oriented product line, where the features can be mapped to the workflow or dynamic services through the service analysis. Thus, the variation of the features can be dynamically bound during runtime by means of selecting the services with right quality levels. These works are based on the clear mapping between the features and the program artefacts with a well-designed feature model. However, the solution to the complex mapping problem is not addressed. Our method takes it into consideration and thus proposes the role model to clarify the complex mapping as well as identify the program implementation in a finer-grained level.

## **7 Conclusion and Future Work**

DSPL provides a new paradigm for developing and managing self-adaptive systems by introducing the principles and techniques proposed in SPL engineering. By DSPL, we can use feature models to capture runtime variations (i.e. adaptation points), as well as their dependencies, to provide a high-level business view for adaptations. Similar to product derivation in SPL engineering, DSPL requires some kinds of variability mechanisms to map feature reconfigurations to implementation-level adaptations. In this paper, we propose a feature-oriented method to support runtime variability reconfiguration in DSPLs. In the method, the role model is introduced to clarify the complex mappings between features and implementation elements. Furthermore, a reference implementation framework based on dynamic aspect mechanisms is also proposed to implement runtime reconfigurations. As a result, the runtime adaptation can be realized in a more systematic way, from the feature-level variation reconfiguration, to the role-level interaction reconfiguration, and finally to the implementation-level code adaptation adopting dynamic AOP.

As for our future work, we will mostly concentrate on improving the limitations discussed in 5.3. More concretely, we will take the tool and runtime infrastructure development as the first step in our plan. The tool and infrastructure are anticipated to

support role modeling, feature traceability specification, and automatic code generation for dynamic aspect bindings.

**Acknowledgments.** This work is supported by National Natural Science Foundation of China under Grant No. 90818009.

## References

1. A.G.Ganek and T.A.Corbi. The dawning of the autonomic computing era. In: IBM Systems Journal, 42(1): pp. 5-18 (2003).
2. S.Hallsteinsen, M.Hinchey, S.Park and K.Schmid. Dynamic Software Product Line. In: Computer, 41(4): pp. 93-95 (2008).
3. R.Wolfinger, S.Reiter, D.Dhungana, P.Grunbacher and H.Prahofer. Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. In: International Conference on Composition-Based Software Systems (ICCBSS), pp. 21-30 (2008).
4. P.Trinidad, A.Ruiz-Cortes, J.Pena and D.Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In: International Workshop on Dynamic Software Product Line (DSPL 2007).
5. J.Lee and K.C.Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In: International Software Product Line Conference (SPLC), pp. 131-140 (2006).
6. J.Lee and D.Muthig. Feature-Oriented Analysis and Specification of Dynamic Product Reconfiguration. In: International Conference on Software Reuse (ICSR), pp.154-165(2008).
7. G.Kotonya, J.Lee and D.Robinson. A Consumer-Centred Approach for Service-Oriented Product Line Development. In: Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 211-220 (2009).
8. J.Lee and G.Kotonya. Combining Service-Oriented with Product Line Engineering. In: IEEE Software, pp. 35 – 41 (2010).
9. A.G.J. Jansen, R. Smedinga, J. van Gurp and J. Bosch. First class feature abstractions for product derivation. In IEE Proc.-Softw., Vol. 151, No. 4, (2004).
- 10.K.C.Kang, SG.Cohen, JA.Hess, WE.Novak and AS.Peterson. Feature-oriented domain analysis feasibility study. In: Technical reports, SEI, Carnegie Mellon University, (1990).
- 11.K.C.Kang et al. FORM: A feature-oriented reuse method with domain-specific architecture. In: Annals of Software Engineering, V5: pp. 143-168, (1998).
- 12.Wei Zhang, Hong Mei and Haiyan Zhao. Feature-driven requirement dependency analysis and high-level software design. In: Requirements Eng, Vol.11, pp. 205–220 (2006).
- 13.M.Riebisch and R.Brcina. Optimizing Design for Variability Using Traceability Links. In: International Conference on Engineering of Computer Based Systems, pp. 235-244 (2008).
- 14.C. Bockisch, M. Haupt, M. Mezini and K. Ostermann. Virtual Machine Support for Dynamic Join points. In: International Conference on Aspect-Oriented Software Development, (AOSD 2004).
- 15.D.Khan. JBoss-AOP. (2008) Available at URL: <http://www.jboss.org/jbossaop/>.
- 16.K.R.Dittrich, S.Gatzju and A.Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In: LNCS 985, Springer, pp. 3-20, (1995).
- 17.AspectJ Team. AspectJ Project. <http://www.eclipse.org/aspectj/>.
- 18.X.Peng, L.Shen and W.Zhao. Feature Implementation Modeling based Product Derivation in Software Product Line. In: International Conference on Software Reuse (ICSR), pp. 142-153 (2008).
- 19.L.Shen, X.Peng and W.Zhao. A comprehensive feature-oriented traceability model for software product line development. In: Australian Software Engineering Conference (ASWEC), pp. 210-219 (2009).