e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Aspect-Oriented Change Realizations and Their Interaction

Valentino Vranić*, Radoslav Menkyna*, Michal Bebjak*, Peter Dolog**
*Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Slovakia
**Department of Computer Science, Aalborg University, Denmark
vranic@fiit.stuba.sk, radu@ynet.sk, mbebjak@gmail.com, dolog@cs.aau.dk

Abstract

With aspect-oriented programming, changes can be treated explicitly and directly at the programming language level. An approach to aspect-oriented change realization based on a two-level change type model is presented in this paper. In this approach, aspect-oriented change realizations are mainly based on aspect-oriented design patterns or themselves constitute pattern-like forms in connection to which domain independent change types can be identified. However, it is more convenient to plan changes in a domain specific manner. Domain specific change types can be seen as subtypes of generally applicable change types. These relationships can be maintained in a form of a catalog. Some changes can actually affect existing aspect-oriented change realizations, which can be solved by adapting the existing change implementation or by implementing an aspect-oriented change realization of the existing change without having to modify its source code. As demonstrated partially by the approach evaluation, the problem of change interaction may be avoided to a large extent by using appropriate aspect-oriented development tools, but for a large number of changes, dependencies between them have to be tracked. Constructing partial feature models in which changes are represented by variable features is sufficient to discover indirect change dependencies that may lead to change interaction.

1. Introduction

Change realization consumes enormous effort and time during software evolution. Once implemented, changes get lost in the code. While individual code modifications are usually tracked by a version control tool, the logic of a change as a whole vanishes without a proper support in the programming language itself.

By its capability to separate crosscutting concerns, aspect-oriented programming enables to deal with change explicitly and directly at programming language level. Changes implemented this way are pluggable and — to the great extent — reapplicable to similar applications, such as applications from the same product line.

Customization of web applications represents a prominent example of that kind. In customization, a general application is being adapted to the client's needs by a series of changes. With each new version of the base application, all the changes have to be applied to it. In many occasions, the difference between the new and old application does not affect the structure of changes, so if changes have been implemented using aspect-oriented programming, they can be simply included into the new application build without any additional effort.

Even conventionally realized changes may interact, i.e. they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. This is even more remarkable in aspect-oriented change realization due to pervasiveness of aspect-oriented programming as such.

We have already reported briefly our initial work in change realization using aspect-oriented programming [1]. In this paper[1], we present our improved view of the approach to change realization based on a two-level change type model. Section 2 presents our approach to aspect-oriented change realization. Section 3 describes briefly the change types we have discovered so far in the web application domain. Section 4 discusses how to deal with a change of a change. Section 5 proposes a feature modeling based approach of dealing with change interaction. Section 6 describes the approach evaluation and outlooks for tool support. Section 7 discusses related work. Section 8 presents conclusions and directions of further work.

2. Changes as Crosscutting Requirements

A change is initiated by a change request made by a user or some other stakeholder. Change requests are specified in domain notions similarly as initial requirements are. A change request tends to be focused, but it often consists of several different — though usually interrelated — requirements that specify actual changes to be realized. By decomposing a change request into individual changes and by abstracting the essence out of each such change while generalizing it at the same time, a change type applicable to a range of the applications that belong to the same domain can be defined.

We will present our approach by a series of examples on a common scenario[2]. Suppose a merchant who runs his online music shop purchases a general affiliate marketing software [11] to advertise at third party web sites denoted as affiliates. In a simplified schema of affiliate marketing, a customer visits an affiliate's site which refers him to the merchant's site. When he buys something from the merchant, the provision is given to the affiliate who referred the sale. A general affiliate marketing software enables to manage affiliates, track sales referred by these affiliates, and compute provisions for referred sales. It is also able to send notifications about new sales, signed up affiliates, etc.

The general affiliate marketing software has to be adapted (customized), which involves a series of changes. We will assume the affiliate marketing software is written in Java, so we can use AspectJ, the most popular aspect-oriented language, which is based on Java, to implement some of these changes.

In the AspectJ style of aspect-oriented programming, the crosscutting concerns are captured in units called aspects. Aspects may contain fields and methods much the same way the usual Java classes do, but what makes possible for them to affect other code are genuine aspect-oriented constructs, namely: *pointcuts*, which specify the places in the code to be affected, *advices*, which implement the additional behavior before, after, or instead of the captured *join point* (a well-defined place in the program execution) — most often method calls or executions — and *inter-type declarations*, which enable introduction of new members into types, as well as introduction of compilation warnings and errors.

2.1. Domain Specific Changes

One of the changes of the affiliate marketing software would be adding a backup SMTP server to ensure delivery of the notifications to users. Each time the affiliate marketing software needs to send a notification, it creates an instance of the SMTPServer class which handles the connection to the SMTP server.

An SMTP server is a kind of a resource that needs to be backed up, so in general, the type of the change we are talking about could be denoted as *Introducing Resource Backup*. This change type is still expressed in a domain specific way. We can clearly identify a crosscutting concern of maintaining a backup resource that

[1] This paper represents an extended version of our paper presented at CEE-SET 2008 [28].
[2] This is an adapted scenario published in our earlier work [1].

has to be activated if the original one fails and implement this change in a single aspect without modifying the original code:

```
public class SMTPServerM extends SMTPServer {
 ...
}
 ...
public aspect SMTPServerBackupA {
  public pointcut SMTPServerConstructor(URL url,
                                String user,
                                String password):
    call(SMTPServer.new (..)) && args(url, user,
                                password);
  SMTPServer around(URL url, String user,
                                String password):
    SMTPServerConstructor(url, user, password)
  {
    return getSMTPServerBackup(proceed(url, user,
                                password));
  }
  private SMTPServer
  getSMTPServerBackup(SMTPServer obj)
  {
    if (obj.isConnected()) {
      return obj;
    } else {
      return new SMTPServerM(obj.getUrl(),
                                obj.getUser(),
                                obj.getPassword());
    }
  }
}
```

The **around**() advice captures constructor calls of the SMTPServer class and their arguments. This kind of advice takes complete control over the captured join point and its return clause, which is used in this example to control the type of the SMTP server being returned. The policy is implemented in the getSMTPServerBackup() method: if the original SMTP server can't be connected to, a backup SMTP server class SMTPServerM instance is created and returned.

We can also have another aspect — say SMTPServerBackupB — intended for another application configuration that would implement a different backup policy or simply instantiate a different backup SMTP server.

2.2. Generally Applicable Changes

Looking at this code and leaving aside SMTP servers and resources altogether, we notice that it actually performs a class exchange. This idea can be generalized and domain details abstracted out of it bringing us to the *Class Exchange* change type [1] which is based on the Cuckoo's Egg aspect-oriented design pattern [20]:

```
public class AnotherClass extends MyClass {
 ...
}
 ...
public aspect MyClassSwapper {
  public pointcut myConstructors():
    call(MyClass.new ());
  Object around(): myConstructors()
  {
    return new AnotherClass();
  }
}
```

2.3. Applying a Change Type

It would be beneficial if the developer could get a hint on using the Cuckoo's Egg pattern based on the information that a resource backup had to be introduced. This could be achieved by maintaining a catalog of changes in which each domain specific change type would be defined as a specialization of one or more generally applicable changes.

When determining a change type to be applied, a developer chooses a particular change request, identifies individual changes in it, and determines their type. Figure 1 shows an example situation. Domain specific changes of the D1 and D2 type have been identified in the Change Request 1. From the previously identified and cataloged relationships between change types we would know their generally applicable change types are G1 and G2.

A generally applicable change type can be a kind of an aspect-oriented design pattern (consider G2 and AO Pattern 2). A domain specific change realization can also be complemented by an aspect-oriented design pattern (or several ones), which is expressed by an association between them (consider D1 and AO Pattern 1).

Each generally applicable change has a known domain independent code scheme (G2's code scheme is omitted from the figure). This code scheme has to be adapted to the context

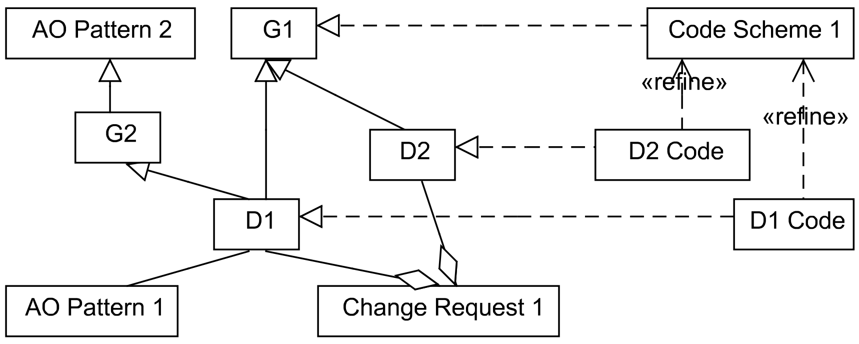

Figure 1. Generally applicable and domain specific changes

of a particular domain specific change, which may be seen as a kind of refinement (consider D1 Code and D2 Code).

3. Catalog of Changes

To support the process of change selection, the catalog of changes is needed in which the generalization–specialization relationships between change types would be explicitly established. The following list sums up these relationships between change types we have identified in the web application domain (the domain specific change type is introduced first):

- One Way Integration: Performing Action After Event,
- Two Way Integration: Performing Action After Event,
- Adding Column to Grid: Performing Action After Event,
- Removing Column from Grid: Method Substitution,
- Altering Column Presentation in Grid: Method Substitution,
- Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification,
- Removing Fields from Form: Additional Return Value Checking/Modification,
- Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event,
- Introducing User Rights Management: Border Control with Method Substitution,
- User Interface Restriction: Additional Return Value Checking/Modifications,
- Introducing Resource Backup: Class Exchange.

We have already described Introducing Resource Backup and the corresponding generally applicable change, Class Exchange. Here, we will briefly describe the rest of the domain specific change types we identified in the web application domain along with the corresponding generally applicable changes. The generally applicable change types are described where they are first mentioned to make sequential reading of this section easier. In a real catalog of changes, each change type would be described separately.

3.1. Integration Changes

Web applications often have to be integrated with other systems. Suppose that in our example the merchant wants to integrate the affiliate marketing software with the third party newsletter which he uses. Every affiliate should be a member of the newsletter. When an affiliate signs up to the affiliate marketing software, he should be signed up to the newsletter, too. Upon deleting his account, the affiliate should be removed from the newsletter, too.

This is a typical example of the *One Way Integration* change type [1]. Its essence is the one way notification: the integrating application notifies the integrated application of relevant events. In our case, such events are the affiliate sign-up and affiliate account deletion.

Such integration corresponds to the *Performing Action After Event* change type [1]. Since events are actually represented by methods, the desired action can be implemented in an after advice:

```
public aspect PerformActionAfterEvent {
  pointcut methodCalls(TargetClass t, int a):...;
  after( /* captured arguments */):
            methodCalls( /* captured arguments */)
  {
            performAction( /* captured arguments */);
  }
  private void performAction( /* arguments */)
  {
    /* action logic */
  }
}
```

The after advice executes after the captured method calls. The actual action is implemented as the performAction() method called by the advice.

To implement the one way integration, in the after advice we will make a post to the newsletter sign-up/sign-out script and pass it the e-mail address and name of the newly signed-up or deleted affiliate. We can seamlessly combine multiple one way integrations to integrate with several systems.

The *Two Way Integration* change type can be seen as a double One Way Integration. A typical example of such a change is data synchronization (e.g., synchronization of user accounts) across multiple systems. When a user changes his profile in one of the systems, these changes should be visible in all of them. In our example, introducing a forum for affiliates with synchronized user accounts for affiliate convenience would represent a Two Way Integration.

3.2. Introducing User Rights Management

In our affiliate marketing application, the marketing is managed by several co-workers with different roles. Therefore, its database has to be updated from an administrator account with limited permissions. A restricted administrator should not be able to decline or delete affiliates, nor modify the advertising campaigns and banners that have been integrated with the web sites of affiliates. This is an instance of the *Introducing User Rights Management* change type.

Suppose all the methods for managing campaigns and banners are located in the campaigns and banners packages. The calls to these methods can be viewed as a region prohibited to the restricted administrator. The Border Control design pattern [20] enables to partition an application into a series of regions implemented as pointcuts that can later be operated on by advices [1]:

```
pointcut prohibitedRegion():
  (within(application.Proxy)
  && call(void *. * (..)))
  || (within(application.campaigns. +)
  && call(void *. * (..)))
  || within(application.banners. +)
  || call(void Affiliate .decline (..))
  || call(void Affiliate .delete (..));
```

What we actually need is to substitute the calls to the methods in the region with our own code that will let the original methods execute only if the current user has sufficient rights. This can be achieved by applying the *Method Substitution* change type which is based on an around advice that enables to change or completely disable the execution of methods. The following pointcut captures all method calls of the method called method() belonging to the TargetClass class:

```
pointcut allmethodCalls(TargetClass t, int a):
  call(ReturnType TargetClass.method(..)) &&
  target(t) && args(a);
```

Note that we capture method calls, not executions, which gives us the flexibility in constraining the method substitution logic by the context of the method call. The **call**() pointcut captures all the calls of TargetClass.method(), the **target**() pointcut is used to capture the reference to the target object, and the method arguments (if we need them) are captured by an **args**() pointcut. In the example code, we assume method() has one integer argument and capture it with this pointcut.

The following example captures the method() calls made within the control flow of any of the CallingClass methods:

```
pointcut specificmethodCalls(TargetClass t, int a):
   call(ReturnType TargetClass.method(a))
   && target(t) && args(a)
   && cflow(call(* CallingClass .*(..)));
```

This embraces the calls made directly in these methods, but also any of the method() calls made further in the methods called directly or indirectly by the CallingClass methods.

By making an around advice on the specified method call capturing pointcut, we can create a new logic of the method to be substituted:

```
public aspect MethodSubstition {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(TargetClass t, int a):
                                          methodCalls(t, a) {
      if (. . .) {
         . . . } // the new method logic
      else
         proceed(t, a);
   }
}
```

3.3. User Interface Restriction

It is quite annoying when a user sees, but can't access some options due to user rights restrictions. This requires a *User Interface Restriction* change type to be applied. We have created a similar situation in our example by a previous change implementation that introduced the restricted administrator (see Sect. 3.2). Since the restricted administrator can't access advertising campaigns and banners, he shouldn't see them in menu either.

Menu items are retrieved by a method and all we have to do to remove the banners and campaigns items is to modify the return value of this method. This may be achieved by applying a *Additional Return Value Checking/Modification* change which checks or modifies a method return value using an around advice:

```
public aspect AdditionalReturnValueProcessing {
   pointcut methodCalls(TargetClass t, int a): . . .;
   private ReturnType retValue;
   ReturnType around():
                  methodCalls(/* captured arguments */) {
      retValue = proceed(/* captured arguments */);
      processOutput(/* captured arguments */);
      return retValue;
   }
   private void processOutput(/* arguments */) {
      // processing logic
   }
}
```

In the around advice, we assign the original return value to the private attribute of the aspect. Afterwards, this value is processed by the processOutput() method and the result is returned by the around advice.

3.4. Grid Display Changes

It is often necessary to modify the way data are displayed or inserted. In web applications, data are often displayed in grids, and data input is usually realized via forms. Grids usually display the content of a database table or collation of data from multiple tables directly. Typical changes required on grid are adding columns, removing them, and modifying their presentation. A grid that is going to be modified must be implemented either as some kind of a reusable component or generated by row and cell processing methods. If the grid is hard coded for a specific view, it is difficult or even impossible to modify it using aspect-oriented techniques.

If the grid is implemented as a data driven component, we just have to modify the data passed to the grid. This corresponds to the Additional Return Value Checking/Modification change (see Sect. 3.3). If the grid is not a data driven component, it has to be provided at least with the methods for processing rows and cells.

Adding Column to Grid can be performed *after an event* of displaying the existing columns of the grid which brings us to the Performing Action After Event change type (see Sect. 3.1). Note that the database has to reflect the change, too. *Removing Column from Grid* requires a conditional execution of the method that displays cells, which may be realized as a Method Substitution change (see Sect. 3.2).

Alterations of a grid are often necessary due to software localization. For example, in Japan and Hungary, in contrast to most other countries, the surname is placed before the given names. The *Altering Column Presentation in Grid* change type requires preprocessing of all the data to be displayed in a grid before actually displaying them. This may be easily achieved by modifying the way the grid cells are rendered, which may be implemented again as a Method Substitution (see Sect. 3.2):

```
public aspect ChangeUserNameDisplay {
   pointcut displayCellCalls(String name, String value):
      call(void UserTable.displayCell (..))  ||
                          args(name, value);
   around(String name, String value):
                          displayCellCalls (name, value) {
      if (name ==
         "<the name of the column to be modified>") {
         . . . // display the modified column
      } else {
         proceed(name, value);
      }
   }
}
```

3.5. Input Form Changes

Similarly to tables, forms are often subject to modifications. Users often want to add or remove fields from forms or pose additional constraints on their input fields. Note that to be possible to modify forms using aspect-oriented programming they may not be hard coded in HTML, but generated by a method. Typically they are generated from a list of fields implemented by an enumeration.

Going back to our example, assume that the merchant wants to know the genre of the music which is promoted by his affiliates. We need to add the genre field to the generic affiliate sign-up form and his profile form to acquire the information about the genre to be promoted at different affiliate web sites. This is a change of the *Adding Fields to Form* type. To display the required information, we need to modify the affiliate table of the merchant panel to display genre in a new column. This can be realized by applying the Enumeration Modification change type to add the genre field along with already mentioned Additional Return Value Checking/Modification in order to modify the list of fields being returned (see Sect. 3.3).

The realization of the *Enumeration Modification* change type depends on the enumeration type implementation. Enumeration types are often represented as classes with a static field for each enumeration value. A single enumeration value type is represented as a class with a field that holds the actual (usually integer) value and its name. We add a new enumeration value by introducing the corresponding static field:

```
public aspect NewEnumType {
   public static EnumValueType
                          EnumType.NEWVALUE =
      new EnumValueType(10, "<new value name>");
}
```

The fields in a form are generated according to the enumeration values. The list of enumeration values is typically accessible via a method provided by it. This method has to be addressed by an Additional Return Value Checking/Modification change.

For *Removing Fields from Form*, an Additional Return Value Checking/Modification change is sufficient. Actually, the enumeration value would still be included in the enumeration, but this would not affect the form generation.

If we want to introduce additional validations on form input fields in an application without a built-in validation, which constitutes an *Introducing Additional Constraint on Fields* change, an *Additional Parameter Checking* change can be applied to methods that process values submitted by the form. This change enables to introduce an additional validation or constraint on method arguments. For this, we have to specify a pointcut that will capture all the calls of the affected methods along with their context similarly as in Sect. 3.2. Their arguments will be checked by the check() method called from within an around advice which will throw WrongParamsException if they are not correct:

```
public aspect AdditionalParameterChecking {
   pointcut methodCalls(TargetClass t, int a): . . .;
   ReturnType around(/* arguments */) throws
                          WrongParamsException:
      methodCalls(/* arguments */) {
      check(/* arguments */);
      return proceed(/* arguments */);
   }
   void check(/* arguments */) throws
                          WrongParamsException {
      if (arg1 != <desired value>)
         throw new WrongParamsException();
   }
}
```

Adding a new validator to an application that already has a built-in validation is realized by simply including it in the list of validators. This can be done by implementing the Performing Action After Event change (see Sect. 3.1), which would add the validator to the list of validators after the list initialization.

4. Changing a Change

Sooner or later there will be a need for a change whose realization will affect some of the already applied changes. There are two possibilities to deal with this situation: a new change can be implemented separately using aspect-oriented programming or the affected change source code could be modified directly. Either way, the changes remain separate from the rest of the application.

The possibility to implement a change of a change using aspect-oriented programming and without modifying the original change is given by the aspect-oriented programming language capabilities. Consider, for example, advices in AspectJ. They are unnamed, so can't be referred to directly. The primitive pointcut **adviceexecution**(), which captures execution of all advices, can be restricted by the **within**() pointcut to a given aspect, but if an aspect contains several advices, advices have to be annotated and accessed by the **@annotation**() pointcut, which was impossible in AspectJ versions that existed before Java was extended with annotations.

An interesting consequence of aspect-oriented change realization is the separation of crosscutting concerns in the application which improves its modularity (and thus makes easier further changes) and may be seen as a kind of aspect-oriented refactoring. For example, in our affiliate marketing application, the integration with a newsletter — identified as a kind of One Way Integration — actually was a separation of integration connection, which may be seen as a concern of its own. Even if these once separated concerns are further maintained by direct source code modification, the important thing is that they remain separate from the rest of the application. Implementing a change of a change using aspect-oriented programming and without modifying the original change is interesting mainly if it leads to separation of another crosscutting concern.

5. Capturing Change Interaction by Feature Models

Some change realizations can *interact*: they may be mutually dependent or some change realizations may depend on the parts of the underlying system affected by other change realizations. With increasing number of changes, change interaction can easily escalate into a serious problem: serious as *feature* interaction.

Change realizations in the sense of the approach presented so far actually resemble features as coherent pieces of functionality. Moreover, they are virtually pluggable and as such represent *variable* features. This brings us to feature modeling as an appropriate technique for managing variability in software development including variability among changes. This section will show how to model aspect-oriented changes using feature modeling.

5.1. Representing Change Realizations

There are several feature modeling notations [26] of which we will stick to a widely accepted and simple Czarnecki–Eisenecker basic notation [5]. Further in this section, we will show how feature modeling can be used to manage change interaction with elements of the notation explained as needed.

Aspect-oriented change realizations can be perceived as variable features that extend an existing system. Fig. 2 shows the change realizations from our affiliate marketing scenario a feature diagram. A feature diagram is commonly represented as a tree whose root represents a concept being modeled. Our concept is our affiliate marketing software. All the changes are modeled as optional features (marked by an empty circle ended edges) that can but do not have to be included in a feature configuration — known also as concept instance — for it to be

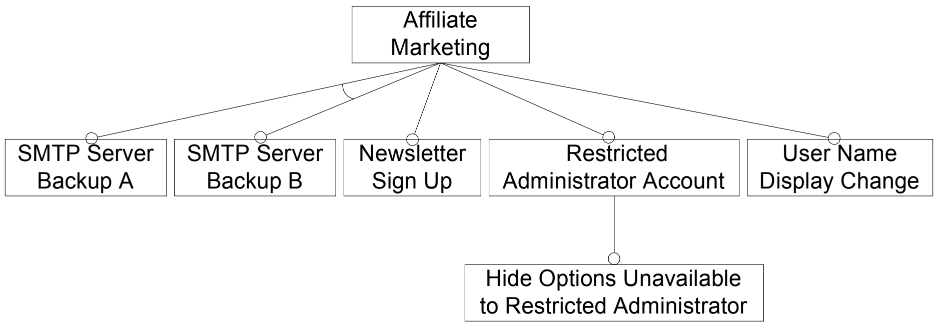

Figure 2. Affiliate marketing software change realizations in a feature diagram

valid. Recall adding a backup SMTP server discussed in Sect. 2.1. We considered a possibility of having another realization of this change, but we don't want both realizations simultaneously. In the feature diagram, this is expressed by alternative features (marked by an arc), so no Affiliate Marketing instance will contain both SMTP Server Backup A and SMTP Server Backup B.

A change realization can be meaningful only in the context of another change realization. In other words, such a change realization requires the other change realization. In our scenario, hiding options unavailable to a restricted administrator makes sense only if we introduced a restricted administrator account (see Sect. 3.3 and 3.2). Thus, the Hide Options Unavailable to Restricted Administrator feature is a subfeature of the Restricted Administrator Account feature. For a subfeature to be included in a concept instance its parent feature must be included, too.

5.2. Identifying Direct Change Interactions

Direct change interactions can be identified in a feature diagram with change realizations modeled as features of the affected software concept. Each dependency among features represents a potential change interaction. A direct change interaction may occur among alternative features or a feature and its subfeatures: such changes may affect the common join points. In our affiliate marketing scenario, alternative SMTP backup server change realizations are an example of such changes. Determining whether changes really interact requires analysis of dependant feature semantics with respect to the implementation of the software being changed. This is beyond feature modeling capabilities.

Indirect feature dependencies may also represent potential change interactions. Additional dependencies among changes can be discovered by exploring the software to which the changes are introduced. For this, it is necessary to have a feature model of the software itself, which is seldom the case. Constructing a complete feature model can be too costly with respect to expected benefits for change interaction identification. However, only a part of the feature model that actually contains edges that connect the features under consideration is needed in order to reveal indirect dependencies among them.

5.3. Partial Feature Model Construction

The process of constructing partial feature model is based on the feature model in which aspect-oriented change realizations are represented by variable features that extend an existing system represented as a concept (see Sect. 5.1).

The concept node in this case is an abstract representation of the underlying software system. Potential dependencies of the change realizations are hidden inside of it. In order to reveal them, we must factor out concrete features from the concept. Starting at the features that represent change realizations (leaves) we proceed bottom up trying to identify their parent features until related changes are not grouped in common subtrees. Figure 3 depicts this process.

The process will be demonstrated on YonBan, a student project management system de-

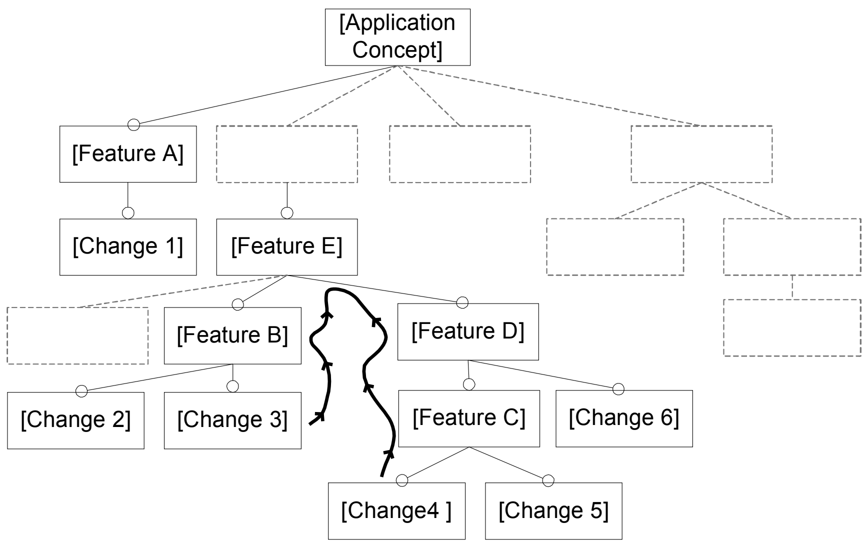

Figure 3. Constructing a partial feature model

veloped at Slovak University of Technology. We will consider the following changes in YonBan and their respective realizations indicated by generally applicable change types:

- Telephone Number Validating (realized as Performing Action After Event): to validate a telephone number the user has entered;
- Telephone Number Formatting (realized as Additional Return Value Checking/Modification): to format a telephone number by adding country prefix;
- Project Registration Statistics (realized as One Way Integration): to gain statistic information about the project registrations;
- Project Registration Constraint (realized as Additional Parameter Checking/Modification): to check whether the student who wants to register a project has a valid e-mail address in his profile;
- Exception Logging (realized as Performing Action After Event): to log the exceptions thrown during the program execution;
- Name Formatting (realized as Method Substitution): to change the way how student names are formatted.

These change realizations are captured in the initial feature diagram presented Fig. 4. Since there was no relevant information about direct dependencies among changes during their specification, there are no direct dependencies among the features that represent them either. The concept of the system as such is marked as open (indicated by square brackets), which means that new variable subfeatures are expected at it. This is so because we show only a part of the analyzed system knowing there are other features there.

Following this initial stage, we attempt to identify parent features of the change realization features as the features of the underlying system that are affected by them. Figure 5 shows such changes identified in our case. We found that Name Formatting affects the Name Entering feature. Project Registration Statistic and Project Registration Constraint change User Registration. Telephone Number Formatting and Telephone Number Validating are changes of Telephone Number Entering. Exception Logging affects all the features in the application, so it remains a direct feature of the concept. All these newly identified features are open because we are aware of the incompleteness of their subfeature sets.

We continue this process until we are able to identify parent features or until all the changes are found in a common subtree of the feature diagram, whichever comes first. In our example, we reached this stage within the following — and thus last — iteration which is presented in Fig. 6: we realized that Telephone Number Entering is a part of User Registration.

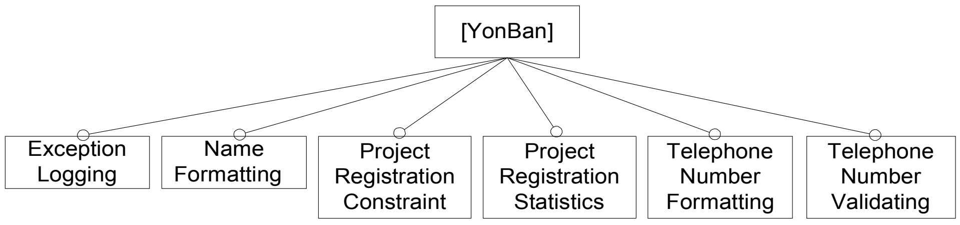
[YonBan]
Exception Logging
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 4. Initial stage of the YonBan partial feature model construction

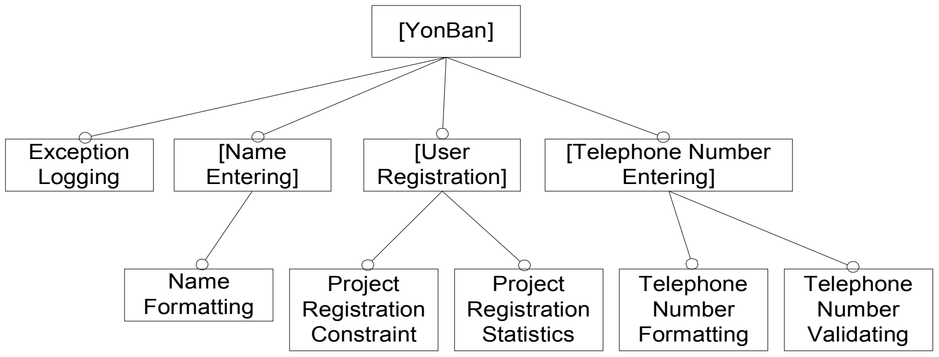
[YonBan]
Exception Logging
[Name Entering]
[User Registration]
[Telephone Number Entering]
Name Formatting
Project Registration Constraint
Project Registration Statistics
Telephone Number Formatting
Telephone Number Validating

Figure 5. Identifying parent features in YonBan partial feature model construction

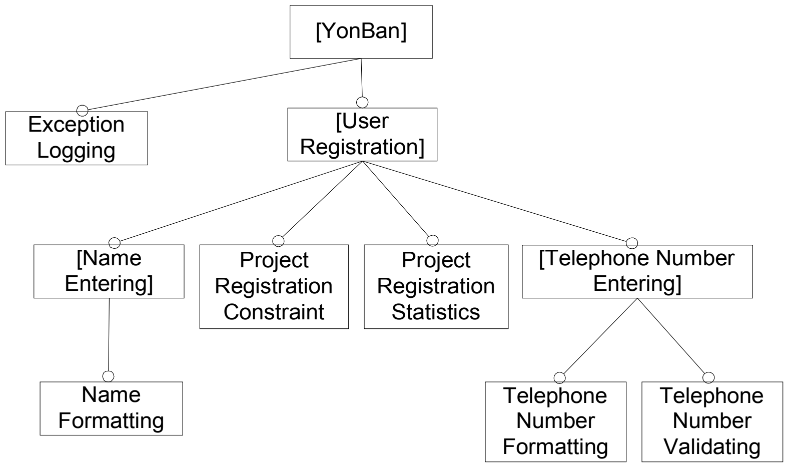
[YonBan]
Exception Logging
[User Registration]
[Name Entering]
Project Registration Constraint
Project Registration Statistics
[Telephone Number Entering]
Name Formatting
Telephone Number Formatting
Telephone Number Validating

Figure 6. The final YonBan partial feature model

5.4. Dependency Evaluation

Dependencies among change realization features in a partial feature model constitute potential change realization interactions. A careful analysis of the feature model can reveal dependencies we have overlooked during its construction.

Sibling features (direct subfeatures of the same parent feature) are potentially interdependent. This problem can occur also among the features that are — to say so — indirect siblings, so we have to analyze these, too. Speaking in terms of change implementation, the code that implements the parent feature altered by one of the sibling change features can be dependent on the code altered by another sibling change feature or vice versa. The feature model points us to the locations of potential interaction.

In our example, we have a partial feature model (recall Fig. 6) and we understand the way the changes should be implemented based on their type (see Sect. 5.3). Project Registration Constraint and Project Registration Statistic change are both direct subfeatures of User Registration. The two aspects that would implement these changes would advise the same project registration method, and this indeed can lead to interaction. In such cases, precedence of aspects should be set (in AspectJ, **dominates** inter-type declaration enables this). Another possible problem in this particular situation is that the Project Registration Constraint change can disable the execution of the project registration method. If the Project Registration Statistic change would use an **execution**() pointcut, everything would be all right. On the other hand, if the Project Registration Statistic change would use a **call**() pointcut, the registration statistic advice would be still executed even when the registration method would not be executed. This would cause an undesirable system behavior where also registrations canceled by Project Registration Constraint would be counted in statistic. The probability of a mistake when a **call**() pointcut is used instead of the **execution**() pointcut is higher if the Project Registration Statistic change would be added first.

Telephone Number Formatting and Telephone Number Validating are another example of direct subfeatures. In this case, the aspects that would implement these changes apply to different join points, so apparently, no interaction should occur. However, a detailed look uncovers that Telephone Number Formatting change alters the value which the Telephone Number Validating change has to validate. This introduces a kind of logical dependency and to this point the two changes interact. For instance, altering Telephone Number Formatting to format the number in a different way may require adapting Telephone Number Validating.

We saw that the dependencies between changes could be as complex as feature dependencies in feature modeling and accordingly represented by feature diagrams. For dependencies appearing among features without a common parent, additional constraints expressed as logical expressions [27] could be used. These constraints can be partly embedded into feature diagrams by allowing them to be directed acyclic graphs instead of just trees [10].

Some dependencies between changes may exhibit only recommending character, i.e. whether they are expected to be included or not included together, but their application remains meaningful either way. An example of this are features that belong to the same change request. Again, feature modeling can be used to model such dependencies with so-called default dependency rules that may also be represented by logical expressions [27].

6. Evaluation and Tool Support Outlooks

We have successfully applied the aspect-oriented approach to change realization to introduce changes into YonBan, the student project management system discussed in previous section. YonBan is based on J2EE, Spring, Hibernate, and Acegi frameworks. The YonBan architecture is based on the Inversion of Control principle and Model-View-Controller pattern.

We implemented all the changes listed in Sect. 5.3. No original code of the system had to be modified. Except in the case of project registration statistics and project registration constraint, which where well separated from the rest of the code, other changes would require extensive code modifications if they have had been implemented the conventional way.

As we discussed in Sect 5.4, we encountered one change interaction: between the telephone number formatting and validating. These two changes are interrelated — they would probably be part of one change request — so it comes as no surprise they affect the same method. However, no intervention was needed in the actual implementation.

We managed to implement the changes easily even without a dedicated tool, but to cope with a large number of changes, such a tool may become crucial. Even general aspect-oriented programming support tools — usually integrated with development environments — may be of some help in this. AJDT (AspectJ Development

Tools) for Eclipse is a prominent example of such a tool. AJDT shows whether a particular code is affected by advices, the list of join points affected by each advice, and the order of advice execution, which all are important to track when multiple changes affect the same code. Advices that do not affect any join point are reported in compilation warnings, which may help detect pointcuts invalidated by direct modifications of the application base code such as identifier name changes or changes in method arguments.

A dedicated tool could provide a much more sophisticated support. A change implementation can consist of several aspects, classes, and interfaces, commonly denoted as types. The tool should keep a track of all the parts of a change. Some types may be shared among changes, so the tool should enable simple inclusion and exclusion of changes. This is related to change interaction, which can be addressed by feature modeling as we described in the previous section.

7. Related Work

The work presented in this paper is based on our initial efforts related to aspect-oriented change control [8] in which we related our approach to change-based approaches in version control. We concluded that the problem with change-based approaches that could be solved by aspect-oriented programming is the lack of programming language awareness in change realizations.

In our work on the evolution of web applications based on aspect-oriented design patterns and pattern-like forms [1], we reported the fundamentals of aspect-oriented change realizations based on the two level model of domain specific and generally applicable change types, as well as four particular change types: Class Exchange, Performing Action After Event, and One/Two Way Integration.

Applying feature modeling to maintain change dependencies (see Sect. 4) is similar to constraints and preferences proposed in SIO software configuration management system [4]. However, a version model for aspect dependency management [23] with appropriate aspect model that enables to control aspect recursion and stratification [2] would be needed as well.

We tend to regard changes as concerns, which is similar to the approach of facilitating configurability by separation of concerns in the source code [9]. This approach actually enables a kind of aspect-oriented programming on top of a versioning system. Parts of the code that belong to one concern need to be marked manually in the code. This enables to easily plug in or out concerns. However, the major drawback, besides having to manually mark the parts of concerns, is that — unlike in aspect-oriented programming — concerns remain tangled in code.

Others have explored several issues generally related to our work, but none of these works aims at actual capturing changes by aspects. These issues include database schema evolution with aspects [12] or aspect-oriented extensions of business processes and web services with crosscutting concerns of reliability, security, and transactions [3]. Also, an increased changeability of components implemented using aspect-oriented programming [17], [18], [22] and aspect-oriented programming with the frame technology [19], as well as enhanced reusability and evolvability of design patterns achieved by using generic aspect-oriented languages to implement them [24] have been reported. The impact of changes implemented by aspects has been studied using slicing in concern graphs [15].

While we do see potential of aspect-orientation for configuration and reconfiguration of applications, our current work does not aim at automatic adaptation in application evolution, such as event triggered evolutionary actions [21], evolution based on active rules [6], adaptation of languages instead of software systems [16], or as an alternative to version model based context-awareness [7], [13].

8. Conclusions and Further Work

In this paper, we have described our approach to change realization using aspect-oriented programming and proposed a feature modeling based approach of dealing with change interaction. We deal with changes at two levels distinguishing between domain specific and generally applicable change types. We described change types specific to web application domain along with corresponding generally applicable changes. We also discussed consequences of having to implement a change of a change.

The approach does not require exclusiveness in its application: a part of the changes can be realized in a traditional way. In fact, the approach is not appropriate for realization of all changes, and some of them can't be realized by it at all. This is due to a technical limitation given by the capabilities of the underlying aspect-oriented language or framework. Although some work towards addressing method-level constructs such as loops has been reported [14], this is still uncommon practice. What is more important is that relying on the inner details of methods could easily compromise the portability of changes across the versions since the stability of method bodies between versions is questionable.

Change interaction can, of course, be analyzed in code, but it would be very beneficial to deal with it already during modeling. We showed that feature modeling can successfully be applied whereby change realizations would be modeled as variable features of the application concept. Based on such a model, change dependencies could be tracked through feature dependencies. In the absence of a feature model of the application under change, which is often the case, a partial feature model can be developed at far less cost to serve the same purpose.

For further evaluation, it would be interesting to develop catalogs of domain specific change types of other domains like service-oriented architecture for which we have a suitable application developed in Java available [25]. Although the evaluation of the approach has shown the approach can be applied even without a dedicated tool support, we believe that tool support is important in dealing with change interaction, especially if their number is high.

By applying the multi-paradigm design with feature modeling [27] to select the generally applicable changes (understood as paradigms) appropriate to given application specific changes we may avoid the need for catalogs of domain specific change types or we can even use it to develop them. This constitutes the main course of our further research.

Acknowledgements The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

References

[1] M. Bebjak, V. Vranić, and P. Dolog. Evolution of web applications with aspect-oriented design patterns. In M. Brambilla and E. Mendes, editors, *Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007*, pages 80–86, Como, Italy, July 2007.

[2] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In R. Hirschfeld et al., editors, *Proc. of NODe 2006*, LNI P-88, pages 49–64, Erfurt, Germany, Sept. 2006. GI.

[3] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, secure, and transacted web service compositions with AO4BPEL. In *4th IEEE European Conf. on Web Services (ECOWS 2006)*, pages 23–34, Zürich, Switzerland, Dec. 2006. IEEE Computer Society.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[5] K. Czarnecki and U. W. Eisenecker. *Generative Programing: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[6] F. Daniel, M. Matera, and G. Pozzi. Combining conceptual modeling and active rules for the design of adaptive web applications. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[7] F. Dantas, T. Batista, N. Cacho, and A. Garcia. Towards aspect-oriented programming for context-aware systems: A comparative study. In *Proc. of 1st International Workshop on Software Engineering for Pervasive Computing Ap-*

plications, Systems, and Environments, SEPCASE'07, Minneapolis, USA, May 2007. IEEE.

[8] P. Dolog, V. Vranić, and M. Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, Dec. 2001.

[9] Z. Fazekas. Facilitating configurability by separation of concerns in the source code. *Journal of Computing and Information Technology (CIT)*, 13(3):195–210, Sept. 2005.

[10] R. Filkorn and P. Návrat. An approach for integrating analysis patterns and feature diagrams into model driven architecture. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *Proc. 31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, Liptovský Jan, Slovakia, Jan. 2005. Springer.

[11] S. Goldschmidt, S. Junghagen, and U. Harris. *Strategic Affiliate Marketing.* Edward Elgar Publishing, 2003.

[12] R. Green and A. Rashid. An aspect-oriented framework for schema evolution in object-oriented databases. In *Proc. of the Workshop on Aspects, Components and Patterns for Infrastructure Software (in conjunction with AOSD 2002)*, Enschede, Netherlands, Apr. 2002.

[13] M. Grossniklaus and M. C. Norrie. An object-oriented version model for context-aware data management. In M. Weske, M.-S. Hacid, and C. Godart, editors, *Proc. of 8th International Conference on Web Information Systems Engineering, WISE 2007*, LNCS 4831, Nancy, France, Dec. 2007. Springer.

[14] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proc. of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006*, pages 63–74, Bonn, Germany, 2006. ACM.

[15] S. Khan and A. Rashid. Analysing requirements dependencies and change impact using concern slicing. In *Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008)*, Nantes, France, July 2006.

[16] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč. Functional approach to the adaptation of languages instead of software systems. *Computer Science and Information Systems Journal (ComSIS)*, 4(2), Dec. 2007.

[17] A. A. Kvale, J. Li, and R. Conradi. A case study on building COTS-based system using aspect-oriented programming. In *2005 ACM Symposium on Applied Computing*, pages 1491–1497, Santa Fe, New Mexico, USA, 2005. ACM.

[18] J. Li, A. A. Kvale, and R. Conradi. A case study on improving changeability of COTS-based system using aspect-oriented programming. *Journal of Information Science and Engineering*, 22(2):375–390, Mar. 2006.

[19] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects. In *Workshop on Aspects, Componentsand Patterns for Infrastructure Software (held with AOSD 2004, International Conference on Aspect-Oriented Software Development)*, Lancaster, UK, Mar. 2004.

[20] R. Miles. *AspectJ Cookbook.* O'Reilly, 2004.

[21] F. Molina-Ortiz, N. Medina-Medina, and L. García-Cabrera. An author tool based on SEM-HP for the creation and evolution of adaptive hypermedia systems. In *Workshop Proc. of 6th Int. Conf. on Web Engineering (ICWE 2006)*, New York, NY, USA, 2006. ACM Press.

[22] O. Papapetrou and G. A. Papadopoulos. Aspect-oriented programming for a component based real life application: A case study. In *2004 ACM Symposium on Applied Computing*, pages 1554–1558, Nicosia, Cyprus, 2004. ACM.

[23] E. Pulvermüller, A. Speck, and J. O. Coplien. A version model for aspect dependency management. In *Proc. of 3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 70–79, Erfurt, Germany, Sept. 2001. Springer.

[24] T. Rho and G. Kniesel. *Independent evolution of design patterns and application logic with generic aspects — a case study.* Technical Report IAI-TR-2006-4, University of Bonn, Bonn, Germany, Apr. 2006.

[25] V. Rozinajová, M. Braun, P. Návrat, and M. Bieliková. Bridging the gap between service-oriented and object-oriented approach in information systems development. In D. Avison, G. M. Kasper, B. Pernici, I. Ramos, and D. Roode, editors, *Proc. of IFIP 20th World Computer Congress, TC 8, Information Systems*, Milano, Italy, Sept. 2008. Springer Boston.

[26] V. Vranić. Reconciling feature modeling: A feature modeling metamodel. In M. Weske and P. Liggsmeyer, editors, *Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004)*, LNCS

3263, pages 122–137, Erfurt, Germany, Sept. 2004. Springer.

[27] V. Vranić. Multi-paradigm design with feature modeling. *Computer Science and Information Systems Journal (ComSIS)*, 2(1):79–102, June 2005.

[28] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing applications with aspect-oriented change realization. In *Proc. of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008*, LNCS, Brno, Czech Republic, 2008.