Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling

Radoslav Menkyna and Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava,
Ilkovičova 3, 84216 Bratislava 4, Slovakia
Radoslav.Menkyna@softec.sk, vranic@fiit.stuba.sk

Abstract. It has been shown earlier that aspect-oriented change realization based on a two-level change type framework can be employed to deal with changes so they can be realized in a modular, pluggable, and reusable way. In this paper, this idea is extended towards enabling direct change manipulation using multi-paradigm design with feature modeling. For this, generally applicable change types are considered to be (small-scale) paradigms and expressed by feature models. Feature models of the Method Substitution and Performing Action After Event change types are presented as examples. In this form, generally applicable change types enter an adapted process of the transformational analysis to determine their application by their instantiation over an application domain feature model. The application of the transformational analysis in identifying the details of change interaction is presented.
Keywords: change, aspect-oriented programming, multi-paradigm design, feature modeling, change interaction

1 Introduction

Changes of software applications exhibit crosscutting nature either intrinsically by being related to many different parts of the application they affect or by their perception as separate units that can be included or excluded from a particular application build. It is exactly aspect-oriented programming that can provide suitable means to capture this crosscutting nature of changes and to realize them in a pluggable and reapplicable way [1].

Particular mechanisms of aspect-oriented change introduction determine the change type. Some of these change types have already been documented [2,1], so by just identifying the type of the change being requested, we can get a pretty good idea of its realization. This is not an easy thing to do. One possibility is to have a two-level change type model with some change types being close to the application domain and other change types determining the realization, while their mapping is being maintained in a kind of a catalog [1].

But what if such a catalog for a particular domain does not exist? To postpone change realization and develop a whole catalog may be unacceptable with respect to

time and effort needed. The problem of selecting a suitable realizing change type resembles paradigm selection in multi-paradigm design [3]. This other way around – to treat change realization types as paradigms and employ multi-paradigm design to select the appropriate one – is the topic of this paper.

We first take a look at the two-level aspect-oriented change realization model (Sect. 2). Subsequently, the approach to modeling change realization types as paradigms using feature modeling is introduced (Sect. 3). The approach employs the application domain feature model with changes expressed as features (Sect. 4). The key part of the approach is the transformational analysis – the process of finding a suitable paradigm – tailored to change realization (Sect. 5). Afterwards, it is shown how the transformational analysis results can be used to identify change interaction (Sect. 6). The approach is discussed with respect to related work (Sect. 7). Concluding notes close the paper (Sect. 8).

2 Two-Level Change Realization Framework

In our earlier work [2,1], we proposed a two-level aspect-oriented change realization framework. Changes come in the form of change requests each of which may consist of several changes. We understand a change as a requirement focused on a particular issue perceived as indivisible from the application domain perspective.

Given a particular change, a developer determines the domain specific change type that corresponds to it. Domain specific change types represent abstractions and generalizations of changes expressed in the vocabulary of a particular domain. A developer gets a clue to the change realization from the cataloged mappings of domain specific change types to generally applicable change types, which represent abstractions and generalizations of change realizations in a given solution domain (aspect-oriented language or framework). Each generally applicable change type provides an example code of its realization. It can also be a kind of an aspect-oriented design pattern or a domain specific change can even be directly mapped to one or more aspect-oriented design patterns.

As an example, consider some changes in the general affiliate marketing software purchased by a merchant who runs his online music shop to advertise at third party web sites (denoted as affiliates).[1] This software tracks customer clicks on the merchant's commercials (e.g., banners) placed in affiliate sites and whether they led to buying goods from the merchant in which case the affiliate who referred the sale would get the provision.

Consider a change that subsumes the integration of the affiliate marketing software with the third party newsletter used by the merchant so that every affiliate would 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. This is an instance of the change type called One Way Integration [2], one of the web application domain specific change types. Its essence is the one way notification: the integrating application notifies the

[1] This is an extended scenario originally published in our earlier work [2,1].

integrated application of relevant events. In this case, such events are the affiliate sign up and affiliate account deletion.

The catalog of changes [1] would point us to the Performing Action After Event generally applicable change type. As follows from its name, it describes how to implement an action after an event in general. Since events are actually represented by methods, the desired action can be implemented in an after advice [2]:

```
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 newsletter sign up change, 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.

As another example, consider a change is needed to prevent attempts to register without providing an e-mail address. This is actually an instance of the change type called Introducing Additional Constraint on Fields [2], which can be realized using Performing Action After Event or Additional Parameter Checking, but if we assume no form validation mechanism is present, even the most general Method Substitution (which wasn't considered originally [1] for this) can be used to capture method calls:

```
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 Generally Applicable Change Types as Paradigms

Generally applicable change types are independent of the application domain and may even apply to different aspect-oriented languages and frameworks (with an adapted code scheme, of course). The expected number of generally applicable change types that would cover all significant situations is not high. In our experiments, we managed to cope with all situations using only six of them.

On the other hand, in the domain of web applications, eleven application specific changes we identified so far cover it only partially. Each such change type requires a thorough exploration in order to discover all possible realizations by generally applicable change types and design patterns with conditions for their use, and it is not likely that someone would be willing to invest effort into developing a catalog of changes apart of the momentarily needs.

The problem of selecting a suitable generally applicable change type resembles the problem of the selection of a paradigm suitable to implement a particular application domain concept, which is a subject of multi-paradigm approaches [4]. Here, we will consider *multi-paradigm design with feature modeling* (MPD_{FM}), which is based on an adapted Czarnecki-Eisenecker [5] feature modeling notation [6]. Section 3.1 explains how paradigms are modeled in MPD_{FM}. Section 3.2 and 3.3 introduces two examples of change paradigm models.

3.1 Modeling Paradigms

In MPD_{FM}, paradigms are understood as *solution domain concepts* that correspond to programming language mechanisms (like inheritance or class). Such paradigms are being denoted as small-scale to distinguish them from the common concept of the (large-scale) paradigm as a particular approach to programming (like object-oriented or procedural programming) [3].

In MPD_{FM}, feature modeling is used to express paradigms. A feature model consists of a set of feature diagrams, information associated with concepts and features, and constraints and default dependency rules associated with feature diagrams. A feature diagram is usually understood as a directed tree whose root represents a concept being modeled and the rest of the nodes represent its features [7].

The features may be common to all concept instances (feature configurations) or variable, in which case they appear only in some of the concept instances. Features are selected in a process of concept instantiation. Those that have been selected are denoted as bound. The time at which this binding (or choosing not to bind) happens is called binding time. In paradigm modeling, the set of binding times is given by the solution model. In AspectJ we may distinguish among source time, compile time, load time, and runtime.

Each paradigm is considered to be a separate concept and as such presented in its own feature diagram that describes what is common to all paradigm instances (its applications), and what can vary, how it can vary, and when this happens. Consider the AspectJ aspect paradigm feature model shown in Fig. 1. Each aspect is named, which is modeled by a mandatory feature Name (indicated by a filled circle ended edge). The aspect paradigm articulates related structure and behavior that crosscuts otherwise possibly unrelated types. This is modeled by optional features Inter-Type Declarations, Advices, and Pointcuts (indicated by empty circle ended edges). These features represent references to equally named auxiliary concepts that represent plural forms of respective concepts that actually represent paradigms in their own right (and their own feature models [3]). To achieve its intent, an aspect may – similarly to a class – employ Methods (with the method being yet another paradigm) and Fields.

An aspect in AspectJ is instantiated automatically by occurrence of the join points it addresses in accordance with Instantiation Policy. The features that represent different instantiation policies are mandatory alternative features (indicated by an arc over mandatory features), which means that exactly one of them must be selected. An aspect can be Abstract, in which case it can't be instantiated, so it can't have Instantiation Policy either, which is again modeled by mandatory alternative features.

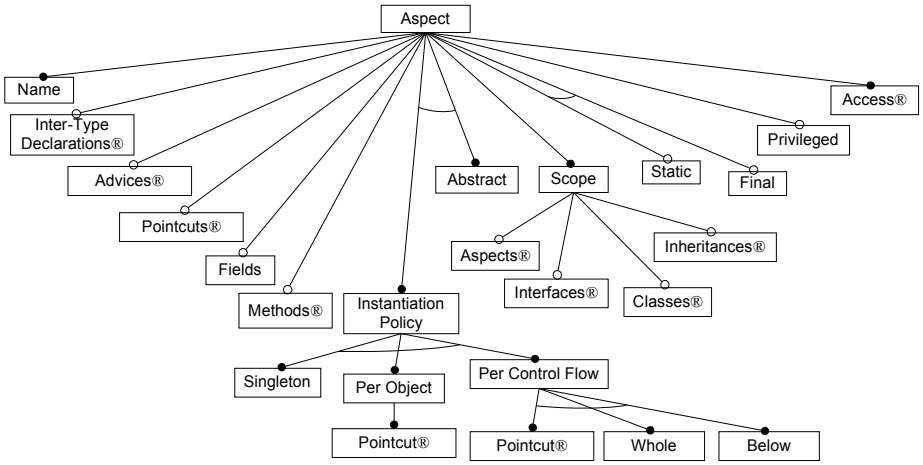


Fig. 1. The AspectJ aspect paradigm (adopted from [3]).

An aspect can be declared to be Static or Final. It doesn't have to be either of the two, but it can't be both, which is modeled by optional alternative features of which only one may be selected (indicated by an arc over optional features). An aspect can also be Privileged over other aspects and it has its type of Access, which is modeled as a reference to a separately expressed auxiliary concept. All the features in the aspect paradigm are bound at source time.

The constraint associated with the aspect paradigm feature diagram means that the aspect is either Final or Abstract. We use first-order predicate logic to express constraints associated with feature diagrams, but OCL could be employed, too, as a widely accepted and powerful notation for such uses (but even of wider applicability, e.g. instead of object algebras [8]).

Generally applicable changes may be seen as a kind of conceptually higher language mechanisms and modeled as paradigms in the sense of MPD_{FM}.

3.2 Method Substitution

Figure 2 shows the Method Substitution change type paradigm model. All the features have source time binding. This change type enables to capture calls to methods (Original Method Calls) with or without the context (Context) and to alter the functionality they implement by the additional functionality it provides (Altering Functionality) which includes the possibility of affecting the arguments (Check/Modify Arguments) or return value (Check/Modify Return Value), or even blocking the functionality of the methods whose calls have been captured altogether (Proceed with Original Methods). Note the Context feature subfeatures. They are or-features, which means at least one them has to be selected.

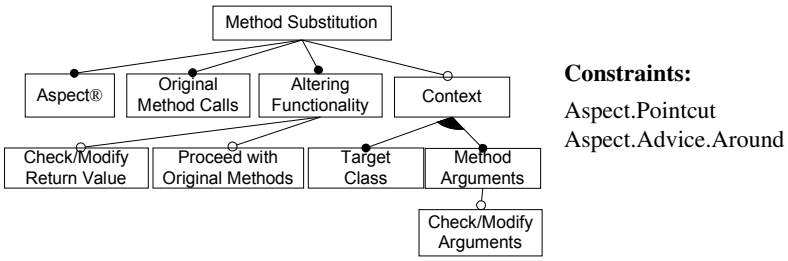


Fig. 2. Method Substitution.

Method Substitution is implemented by an aspect (Aspect) with a pointcut specifying the calls to the methods to be altered by an around advice, which is expressed by the constraints associated with its feature diagram (Fig. 2).

3.3 Performing Action After Event

Figure 3 shows the Performing Action After Event change type paradigm model. All the features have source time binding. This change type is used when an additional action (Action After Event) is needed after some events (Events) of method calls or executions, initialization, field reading or writing, or advice execution (modeled as or-features) taking or not into account their context (Context).

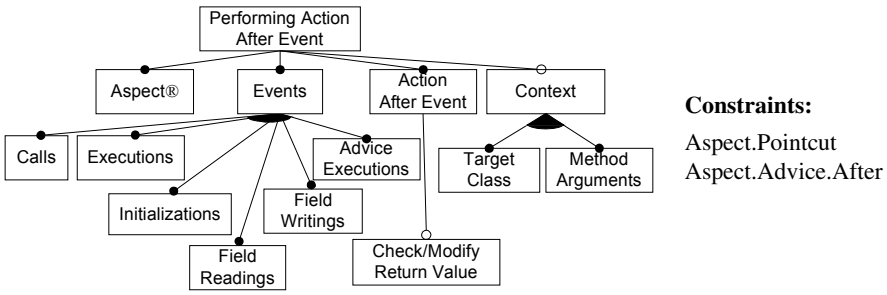


Fig. 3. Performing Action After Event.

Performing Action After Event is implemented by an aspect (Aspect) with a pointcut specifying the events and an after advice over this pointcut used to perform the desired actions, which is expressed by the constraints associated with its feature diagram (Fig. 3).

4 Feature Model of Changes

For the transformational analysis, the application domain feature model that embraces the changes is needed. We will present how changes can be expressed in the application domain feature model in our running example of affiliate tracking software.

4.1 Expressing Changes in a Feature Model

In our affiliate marketing example, we may consider the following changes:

- SMTP Server Backup A/B – to introduce a backup server for sending notifications (with two different implementations, A and B)
- Newsletter Sign Up – to sign up an affiliate to a newsletter when he signs up to the tracking software
- Account Registration Constraint – to check whether the affiliate who wants to register submitted a valid e-mail address
- Restricted Administrator Account – to create an account with a restriction of using some resources
- Hide Options Unavailable to Restricted Administrator – to restrict the user interface
- User Name Display Change — to adapt the order of displaying the first name and surname
- Account Registration Statistics – to gain statistical information about the affiliate registrations

These changes are captured in the initial feature diagram presented in Fig. 4. The concept we model is our affiliate marketing software.[2] All the changes are modeled as optional features as they can, but don't have to be applied. We may consider the possibility of having different realizations of a change of which only one may be applied. This is expressed by alternative features. In the example, no *Affiliate Marketing* instance can contain both *SMTP Server Backup A* and *SMTP Server Backup B*.

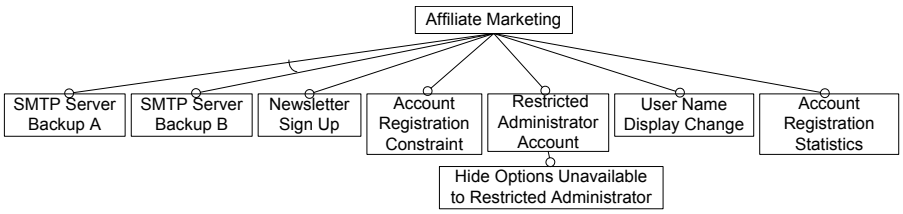


Fig. 4. Changes in the affiliate marketing software.

Some change realizations make sense only in the context of some other change realizations. In other words, such change realization require the other change realizations. In our scenario, hiding options unavailable to a restricted administrator makes

[2] In general, there may be several top-level concepts in one application domain.

sense only if we have introduced a restricted administrator account. This is modeled by having Hide Options Unavailable to Restricted Administrator to be a subfeature of Restricted Administrator Account. For a subfeature to be included in a concept instance, its parent feature must be included, too.

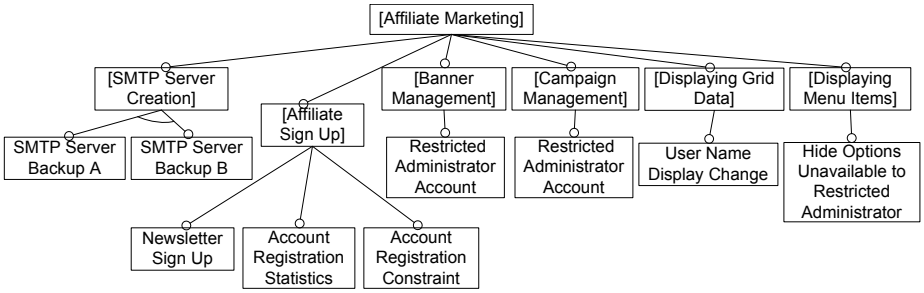
The feature – subfeature relationship represents a direct dependency between two features. Such dependency can be an indication of a possible interaction between change realizations. However, with alternative features, no interaction can occur because an application instance can contain only one change realization.

4.2 Partial Feature Model

Often, no feature model of the system is available. Creating the feature model of the whole system is difficult and time consuming. Fortunately, as it has been shown [9] – for the purpose of change interaction analysis, it is a partial feature model is sufficient. The process of constructing a partial feature model starts with the feature model in which aspect-oriented change realizations are represented by variable features that extend the existing system represented by a concept node as an abstract representation of the underlying software system, which is exactly the model we discussed in the previous section.

In partial feature model construction, only the features that potentially take part in change interaction are being identified and modeled. Starting at change features, we proceed bottom up identifying their parent features until related features become grouped in common subtrees [9].

A partial feature model constructed from the initial feature model of the changes being introduced into our affiliate marketing software (presented in Fig. 4) is depicted in Fig. 5. All the identified change parent features are open because the sets of their subfeatures are incomplete, since we model only the changes that affect them, and since there may be other changes in the future.



Constraints:

Hide Operations Unavailable to Restricted Administrator ⇒
Restricted Administration Account

Fig. 5. A partial feature model of the affiliate marketing software.

At this stage, it is possible to identify potential locations of interaction. Such locations are represented as features of the system to which changes are introduced. The highest probability of interaction is among sibling features (direct subfeatures of the same parent feature) because they are potentially interdependent. This is caused by the fact that changes represented by such features usually employ the same or similar pointcuts which is generally a source of unwanted interaction. Such locations should represent primary targets of evaluation during the transformational analysis, which is the topic of the following section.

Interaction can occur also between indirect siblings or non-sibling features. However, with an increasing distance between features that represent changes, the probability of their interaction decreases.

5 Transformational Analysis

The input to the transformational analysis in multi-paradigm design with feature modeling [3] are two feature models: the application domain one and the solution domain one. The output of the transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features that define the code skeleton.

A concept instance is defined as follows [3]:

> An instance *I* of the concept *C* at time *t* is a *C*'s specialization achieved by configuring its features which includes the *C*'s concept node and in which each feature whose parent is included in *I* obeys the following conditions:
>
> 1. All the mandatory features are included in *I*.
> 2. Each variable feature whose binding time is earlier than or equal to *t* is included or excluded in *I* according to the constraints of the feature diagram and those associated with it. If included, it becomes mandatory for *I*.
> 3. The rest of the features, i.e. the variable features whose binding time is later than *t*, may be included in *I* as variable features or excluded according to the constraints of the feature diagram and those associated with it. The constraints (both feature diagram and associated ones) on the included features may be changed as long as the set of concept instances available at later instantiation times is preserved or reduced.
> 4. The constraints associated with *C*'s feature diagram become associated with the *I*'s feature diagram.

5.1 Transformational Analysis of Changes

For determining change types that correspond to the changes that have to be realized, a simplified transformational analysis can be used. Changes presented in the application domain feature model are considered to be application domain concepts, and generally applicable change types to be paradigms. A complete application domain feature model may be used if available, otherwise a partial feature model has to be constructed. For each change *C* from the application domain feature model, the following steps are performed:

1. Select a generally applicable change type *P* that has not been considered for *C* yet.
2. If there are no more paradigms to select, the process for *C* has failed.
3. Try to instantiate *P* over *C* at source time. If this couldn't be performed or if *P*'s root doesn't match with *C*'s root, go to step 1. Otherwise, record the paradigm instance created.

Paradigm instantiation over application domain concepts means that the inclusion of some of the paradigm nodes is being stipulated by the mapping of the nodes of one or more application domain concepts to them in order to ensure the paradigm instances correspond to these application domain concepts.

If the transformational analysis fails for some change, this change is probably an instance of a new change type. The process should continue with AspectJ paradigms, which is the subject of the general transformational analysis [3].

5.2 Example

We will demonstrate the transformational analysis on several changes in the affiliate marketing software (introduced in Sect. 4.1) with the AspectJ paradigm model [3] extended by feature models of the generally applicable change types (see Sect. 3) as a solution domain.

The Restricted Administrator Account change provides an additional check of access rights upon execution of specified methods. Methods should be executed only if access is granted. This scenario suites best to the Method Substitution change type which can control the execution of selected methods, and ensure displaying an error message or logging in case of an access violation event.

Figure 6 shows the transformational analysis of the Restricted Administrator Account change. The Target Class and Method Arguments features are included to capture additional context which is needed by the Proceed with Original Methods feature when the access is granted. The If Access Granted annotation indicates the condition of proceeding with the original methods. Note that the Banner Management and Campaign Management features are mapped to the Original Method Calls feature expressed by an annotation. This means that the change affects the behavior represented by them. Such annotations are crucial to change interaction evaluation (discussed in the next section).

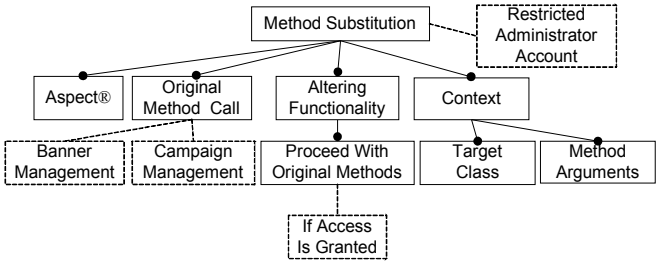


Fig. 6. Transformational analysis of the Restricted User Account change.

The transformational analysis of Account Registration Constraint would be similar. Again, we would employ the Method Substitution change type. The Original Method Calls feature would map to the Affiliate Sign Up feature and the original method will be executed only if a valid e-mail address is provided.

Figure 7 shows the transformational analysis of the Newsletter Sign Up change. Recall that this change adds a new affiliate to the existing list of newsletter recipients, which can be best realized as Performing Action After Event. In this case, the Events feature is mapped to the Affiliate Sign Up feature which represents the execution of the affiliate sign up method. Through Method Arguments, the data about the affiliate being added can be accessed (Affiliate Data) from which his e-mail address can be retrieved and subsequently added to the newsletter recipient list by the Action After Events feature. A similar transformation would apply to the Account Registration Statistics change.

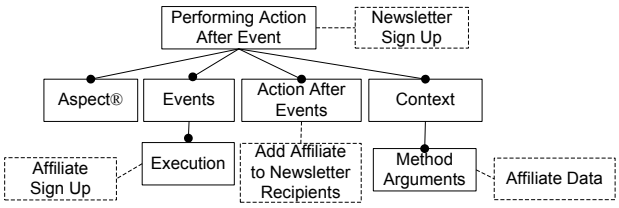


Fig. 7. Transformational analysis of the Newsletter Sign Up change.

6 Change Interaction

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 [1]. The interaction is most probable if multiple changes affect the same functionality. As has been shown, such situations could be identified in part already during the creation of a partial feature model [9], but the transformational analysis can reveal more details needed to avoid the interaction of change realizations.

Consider, for example, the Newsletter Sign Up and Account Registration Statistics changes. Despite they share the target functionality (Affiliate Sign Up), no interaction occurs. This is because both changes are realized using the Performing Action After Event change type which employs an **after**() advice. In such a situation, it is important to check whether the execution order of the advices is significant. In this particular case, it is not.

The Account Registration Constraint change represents a potential source of interaction with Newsletter Sign Up and Account Registration Statistics because it also targets the same functionality. This change is realized using the Method Substitution paradigm through which it can disable the execution of the method that registers a new

affiliate. If the Newsletter Sign Up and Account Registration Statistics change realizations rely on method executions, not calls, i.e. they employ an **execution**() pointcut, no interaction occurs. On the other hand, if the realizations of these changes would rely on method calls, i.e. they would employ a **call**() pointcut, their advices would be executed even if the registration method haven't been executed, which is an undesirable system behavior.

In most cases, the interaction can be solved by adapting change realizations. Unsolvable change interaction should be introduced in the application domain model by constraints that will prevent affected changes from occurring together.

7 Related Work

The impact of changes implemented by aspects has been studied using slicing in concern slice dependency graphs [10]. It has been shown that the application domain feature model can be derived from concern slice dependency graphs [11]. Concern slice dependency graphs provide in part also a dynamic view of change interaction that could be expressed using a dedicated notation (such as UML state machine or activity diagrams) and provided along with the feature model covering the structural view.

Applying program slicing to features implemented as aspects with interaction understood as a slice intersection has been applied so far only to a very simplified version of AspectJ. Extension to cover complicated constructs has been identified as problematic. Even at this simplified level, it appears to be too coarse for applications in which the behavior is embedded in data structures [12].

Even if the original application haven't been a part of a product line, changes modeled as its features tend to form a kind of a product line out of it. This could be seen as a kind of evolutionary development of a new product line [13].

As an alternative to our transformational analysis, framed aspects [14,15] can be applied to the application domain feature model with each change maintained in its own frame in order to keep it separate.

Annotations that determine the feature implementation in so-called *crosscutting feature models* [16] are similar to annotations used in our transformational analysis, but no formal process to determine them is provided.

An approach to introduce program changes by changing the interpreter instead based on grammar weaving has been reported [17]. With respect to suitability of aspect-oriented approach to deal with changes, it is worth mentioning that weaving – a prominent characteristic of aspect-oriented programming – has been identified as crucial for the automation of multi-paradigm software evolution [18].

8 Conclusions and Further Work

The work reported here is a part of our ongoing efforts of comprehensively covering aspect-oriented change realization whose aim is to enable change realization in a modular, pluggable, and reusable way. In this paper, we extended the original idea of having two-level change type framework to facilitate easier aspect-oriented change realization

by enabling direct change manipulation using multi-paradigm design with feature modeling (MPD_{FM}) with generally applicable change types as (small-scale) paradigms.

We introduced the paradigm models of the Method Substitution and Performing Action After Event change types. We also developed paradigm models of other generally applicable change types not presented in this paper such as Enumeration Modification with Additional Return Value Checking/Modification, Additional Return Value Checking/Modification, Additional Parameter Checking or Performing Action After Event, and Class Exchange.

We adapted the process of the general transformational analysis in MPD_{FM} to work with changes as application domain concepts and generally applicable change types as paradigms. We demonstrated how such transformational analysis can help in identifying the details of change interaction.

Our further work includes extending our approach to cover the changes realized by a collaboration of multiple generally applicable change types and design patterns. We also work on improving change type models by expressing them in the Theme notation of aspect-oriented analysis and design [19].

Acknowledgements

The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09 and SOFTEC, s. r. o., Bratislava, Slovakia.

References

1. Vranić, V., Bebjak, M., Menkyna, R., Dolog, P.: Developing Applications with Aspect-oriented Change Realization. In: Proc. of the 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008. LNCS, Brno, Czech Republic, Springer, Heidelberg (2008) (to appear)
2. Bebjak, M., Vranić, V., Dolog, P.: Evolution of Web Applications with Aspect-oriented Design Patterns. In: Brambilla, M., Mendes, E. (eds.) 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, pp. 80–86. Como, Italy (2007)
3. Vranić, V.: Multi-paradigm Design with Feature Modeling. Computer Science and Information Systems Journal (ComSIS) 2(1), 79–102 (2005)
4. Vranić, V.: Towards Multi-paradigm Software Development. Journal of Computing and Information Technology (CIT) 10(2), 133–147 (2002)
5. Czarnecki, K., Eisenecker, U.W.: Generative Programing: Methods, Tools, and Applications. Addison-Wesley (2000)
6. Vranić, V.: Reconciling Feature Modeling: A Feature Modeling Metamodel. In: Weske, M., Liggsmeyer, P. (eds.) Proc. of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004). LNCS, vol. 3263, pp. 122–137. Springer, Heidelberg (2004)
7. Vranić, V., Šípka, M.: Binding Time Based Concept Instantiation in Feature Modeling. In: Morisio, M. (ed.) Proc. of the 9th International Conference on Software Reuse (ICSR 2006). LNCS, vol. 4039, pp. 407–410. Turin, Italy, Springer, Heidelberg (2006)

8. Navarčik, M., Polášek, I.: Object Model Notation. In: Proc. of the 8th International Conference on Information Systems Implementation and Modelling, ISIM 2005, Rožnov pod Radhoštěm, Czech Republic (2005)
9. Vranić, V., Menkyna, R., Bebjak, M., Dolog, P.: Aspect-oriented Change Realizations and their Interaction Submitted to e-Informatica Software Engineering Journal, CEE-SET 2008 special issue.
10. Khan, S., Rashid, A.: Analysing Requirements Dependencies and Change Impact Using Concern Slicing. In: Proc. of Aspects, Dependencies, and Interactions Workshop (affiliated to ECOOP 2008), Nantes, France (2006)
11. Menkyna, R.: Dealing with Interaction of Aspect-oriented Change Realizations Using Feature Modeling. In:Bieliková, M. (ed.) Proc. of the 5th Student Research Conference in Informatics and Information Technologies , IIT.SRC 2009, Bratislava, Slovakia (2009)
12. Monga, M., Beltagui, F., Blair, L.: Investigating Feature Interactions by Exploiting Aspect Oriented Programming. Technical Report comp-002-2003, Lancaster University, Lancaster, UK (2003) Available at http://www.comp.lancs.ac.uk/computing/aose.
13. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley (2000)
14. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: 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 (2004)
15. Loughran, N., Sampaio, A., Rashid, A.: From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. In: MDD for Software Product-lines: Fact or Fiction?, a Workshop held with ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005), Montego Bay, Jamaica (2005)
16. Kulesza, U., Garcia, A., Bleasby, F., Lucena, C.: Instantiating and Customizing Aspect-oriented Architectures Using Crosscutting Feature Models. In: Workshop on Early Aspects held with OOPSLA 2005, San Diego, USA (2005) Available at http://www.early-aspects.net/oopsla05ws.
17. Forgáč, M., Kollár, J.: Adaptive Approach for Language Modification. Journal of Computer Science and Control Systems 2(1), 9–12 (2009)
18. Kollár, J., Porubän, J., Václavík, P., Tóth, M., Bandáková, J., Forgáč, M.: Multi-paradigm Approaches to Systems Evolution. In: Computer Science and Technology Research Survey, Košice, Slovakia (2007)
19. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)