J Ambient Intell Human Comput manuscript No.
(will be inserted by the editor)

An Aspect-Oriented Language for Feature-Modeling

Qinglei Zhang · Ridha Khedri · Jason Jaskolka

Received: date / Accepted: date

Abstract When modeling families of ambient systems, we experience a number of special challenges due to unpredictable variability in the environments of the systems. One solution to deal with these challenges is to adapt aspect-oriented technology to product family modeling.

In this paper, we propose a new language AO-PFA, which adapts the aspect-oriented paradigm to product families. This paradigm enhances the adaptability and evolvability of product families. The proposed language is an extension of the specification language PFA (Product Family Algebra). We discuss the constructs of the proposed language as well as its usage to specify aspects.

Keywords Feature-Modeling · Product Family Algebra · Aspect-Oriented Software Development · Early Aspects · Specification Language · Ambient Systems

1 Introduction and Motivation

Ambient systems involve a multitude of interconnected heterogeneous features that supply end users with a variety of data and functionality. Their stability is contingent on requirements that can cope with high variability (Frei et al, 2010, 2012; Habib and Marimuthu, 2011). Due to the complexity of the hardware and software involved in collecting and acting on the significant amount of data from the environment, there exist a number of special challenges to the feature-modeling process of ambient systems. Where there is a variety in hardware, there is a variety of possible technologies, leading to a collection of predictable variabilities in the product families of similar systems. This observation has been pointed to by Parnas as early as 1976 (Parnas, 1976). Product family modeling was proposed to deal with this problem of handling predictable variabilities. It proposed the simultaneous development of a family of products, rather than of one product at a time. Approaches for handling predictable variability are abundant in a vast literature of feature-modeling techniques (Czarnecki, 1998; Eriksson et al, 2005; Griss et al, 1998; Kang et al, 1990; Riebisch et al, 2002). However, approaches for handling unpredictable variability are somewhat limited.

Unpredictable variabilities can best be illustrated with a security related situation. When considering security, the overall security risk may be strongly affected by changes in only a few subsystems while the system changes and evolves (Solhaug and Seehusen, 2013). Suppose that we need to remotely communicate with an ambient system. Quite often, we would use an authentication feature that is in charge of identifying the caller agent to prevent an intruder from taking control of the system. After some time, suppose we find that there is a flaw in the authentication feature due to the presence of other features. For instance, it is induced by the feature-interaction of the authentication feature with several other features. The question then becomes how can we quickly amend the current feature model to ensure that all the product families involving the identified configuration of features gets amended to replace the flawed configuration by another configuration of features to remedy the situation. Such changes to the configuration of features due to the security issue cannot be predicted

This article is a revised and enlarged version of (Zhang et al, 2012a).

Qinglei Zhang · Ridha Khedri · Jason Jaskolka
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
E-mail: {zhangq33, khedri, jaskolj}@mcmaster.ca

at the time of the feature-modeling of the family. Moreover, it is possible that when the flaw is revealed, several systems may already be deployed in their environment. Usually, modularity based on the separation of concerns in designing systems helps in easing the maintainability of these systems. However, some concerns are inherently distributed over and intertwined with other concerns, and therefore resist such modularisation by conventional approaches. As illustrated in (Nygard et al, 2010), multi-agent systems, which are examples of ambient systems, are associated with many crosscutting concerns such as autonomy, communication, mobility, and security. Because crosscutting concerns frequently hinder the maintainability and modifiability of software qualities, they often render ambient systems difficult to be adapted and evolved. The detection of a defect in a product family leads to the introduction or removal of features in the family or to the amendment of the existing variability by confining it to some products but not others. In order to address the problems engendered by unpredicted changes to features that are related to crosscutting concerns, we can turn to aspect-oriented software development, which provides a powerful way to handle crosscutting concerns in ambient systems.

In the area of software engineering, similar problems are faced at the programming level and are dealt with using aspect-oriented techniques. However, at the programming level, aspect-oriented techniques have shown very mixed results. Aspect-oriented programming leads to systems with high modifiability, but at the same time reduces system performance (Kuusela and Tuominen, 2009). Furthermore, the complexity of the programming languages compared to that of the language we are using at the feature-modeling level makes the aspect weaving process very convoluted and prone to several aspectual compositional problems. At the feature-modeling level, these problems are very minimal (Zhang et al, 2012b). For this reason, we conjecture that aspect-oriented techniques, while they exhibited mixed results at the programming level, can be helpful at the feature-modeling level.

In this paper, we propose a language for the systematic amendment of product families in order to diligently deal with unpredictable changes as soon as they are revealed. This paper builds on the work presented in (Höfner et al, 2006, 2008, 2009, 2011) by expanding the language of *PFA (Product Family Algebra)* to an aspect-oriented language.

This paper is organised as follows. In Section 2, we provide the required background and introduce an example of an elevator product family which is used throughout the paper. In Section 3, we present the proposed specification language. In Section 4, we further illustrate the flexibility of the proposed language by several case studies. In Section 5, we give the theoretical remarks on the adopted weaving procedure and discuss the applicability of the proposed language. In Section 6, we discuss related work reported in the literature of aspect-oriented software development and product family engineering. Lastly, in Section 7, we conclude and point to the highlights of our current and future work.

2 Background

2.1 Product Family Algebra

Product family algebra extends the mathematical notions of semirings to describe and manipulate product families. A semiring is an algebraic structure consisting of a set S with a commutative and associative binary operator $+$ and an associative binary operator $\cdot$. An element $0 \in S$ is the identity element with respect to $+$, while an element $1 \in S$ is the identity element in S with respect to $\cdot$. In addition, operator $\cdot$ distributes over operator $+$ and element 0 annihilates S with respect to $\cdot$. We say a semiring is commutative if the operator $\cdot$ is commutative and a semiring is idempotent if the operator $+$ is idempotent.

Definition 1 (e.g., (Höfner et al, 2009)) A product family algebra is a commutative idempotent semiring $(S, +, \cdot, 0, 1)$, where each element of the semiring is a product family.

In the context of product family modeling, the operators are interpreted as follows:

(a) $+$ denotes the choice between two product families;
(b) $\cdot$ indicates a mandatory composition of two product families;
(c) 0 represents an empty product family;
(d) 1 represents a product family consisting of only a pseudo-product which has no features.

With these interpretations, all other concepts in product family modeling can be expressed mathematically. In particular, optional features can be interpreted as a choice between the features and the pseudo-product 1. For example, a computer product family consists of hardware and software. With regard to hardware, a basic computer is built with a hard disk and a screen, whereas a printer may be added as required (i.e., it is optional). With regard to software, corresponding drivers for each type of hardware component should be provided. The product family of software is specified within the language of product family algebra as follows:

$$\mathrm{sw} = \mathrm{hd\_drv} \cdot \mathrm{scr\_drv} \cdot (1 + \mathrm{prn\_drv}).$$

The notions of *subfamily*, *refinement*, and *constraint* which are introduced in (Höfner et al, 2009) are needed for the rest of the paper. For elements a and b in a product family algebra, the subfamily relation ($\leq$) is defined as $a \leq b \iff_{df} a + b = b$. The subfamily relation indicates that for two given product families a and b, a is a subfamily of b if and only if all the products of a are also products of b. For elements a and b in a product family algebra, the refinement relation ($\sqsubseteq$) is defined as $a \sqsubseteq b \iff_{df} \exists(c \mid: a \leq b \cdot c)$. The refinement relation indicates that for two given product families a and b, a is a refinement of b if and only if every product in family a has at least all the features of some products in family b. For elements a, b, c, d and a product p in product family algebra, the requirement relation ($\rightarrow$) is defined in a family-induction style as:

$$a \xrightarrow{p} b \iff_{df} p \sqsubseteq a \Longrightarrow p \sqsubseteq b$$
$$a \xrightarrow{c+d} b \iff_{df} a \xrightarrow{c} b \wedge a \xrightarrow{d} b$$

The requirement relation is used to specify constraints on product families. For elements a, b and c, $a \xrightarrow{c} b$ can be read as "a requires b within c".

A tool called *Jory* (Alturki and Khedri, 2010), is based on product family algebra and is used to represent and manipulate product families. *Jory* uses a specification language called *PFA (Product Family Algebra)*. The grammar of PFA is given in Figure 1, where '\n' denotes the end of the line. In PFA, there are three types of specification constructs: basic feature declarations, labelled product families, and constraints. Each basic feature is declared with a *basic feature label* preceded by the keyword *bf*. Each product family is defined as an equation with a *product family label* at the left side and a product family algebra term at the right side. A constraint is represented by a triple preceded by the keyword *constraint* and corresponds to a requirement relation in product family algebra. In (Höfner et al, 2006, 2009, 2011), the reader can find more details on the use of this mathematical framework to specify product families.

⟨PFASpec⟩ := (⟨Basic_Feature⟩ | %⟨comment_txt⟩\n)$^+$
(⟨Labelled_Family⟩ | %⟨comment_txt⟩\n)$^+$
(⟨Constraint⟩ | %⟨comment_txt⟩\n)*
⟨Basic_Feature⟩:=*bf* ⟨base_feature_id⟩%⟨comment_txt⟩\n
⟨Labelled_Family⟩:=⟨family_id⟩ =⟨Family_Term⟩
%⟨comment_txt⟩\n
⟨Constraint⟩:=*constraint*(⟨Family_Term⟩, ⟨Family_Term⟩,
⟨Family_Term⟩)%⟨comment_txt⟩\n
⟨Family_Term⟩:=0 | 1 | ⟨base_feature_id⟩ | ⟨family_id⟩
| ⟨Family_Term⟩ + ⟨Family_Term⟩
| ⟨Family_Term⟩ · ⟨Family_Term⟩
⟨base_feature_id⟩:=*String of letters, numbers and "_"*
⟨family_id⟩:=*String of letters, numbers and "_"*
⟨comment_txt⟩:=*String of letters, numbers, symbols*
and space.

Fig. 1: PFA Specification Grammar given in BNF notation

2.2 Aspect-Orientation: Basic Concepts

Aspects are introduced to explicitly encapsulate and implement crosscutting concerns in one module. At different software development stages, the meanings of aspects vary in accordance with the granularities of the concern abstractions. For example, aspects derived from the requirement level could be a non-functional requirement such as quality of service or a functional requirement such as a business rule. Aspects derived from the low implementation level could be caching and buffering. Aspects at analysis and design levels are sometimes referred to as early aspects. Nevertheless, several terminologies are widely and commonly used by the community of aspect-oriented software engineering. First, a *join-point* refers to a point in the execution of the base program where an aspect could be introduced. A *point-cut* selects a set of join-points where a certain aspect should be positioned. An *advice* defines the amendment which should be introduced at the selected join-points. Lastly, *weaving* is the process of combining aspects with a base program. In essence, the point-cut identifies join-points where an aspect should be introduced, while the advice defines the specification of the crosscutting concern.

Without loss of generality, we use an example, given in Figure 2, to illustrate the above concepts and the general mechanism of aspect-oriented programming. The base program in the example is a class type *point*, while the aspect is related to *logging* operations. The point-cut of the logging aspect selects two join-points (underlined instructions in Figure 2) in the base code, while the advice of the aspect introduces the additional *print* operations after those selected joint-points. The code at the right of Figure 2 shows the result of weaving the aspect to the base program.

2.3 Illustrative Example

We use a simplified elevator system as a running example to illustrate the background related to the PFA language and to exemplify the usage of the proposed language. An elevator product family is composed of

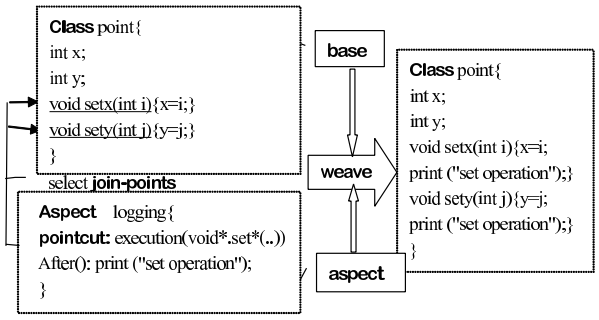

Fig. 2: General aspect-orientation mechanism

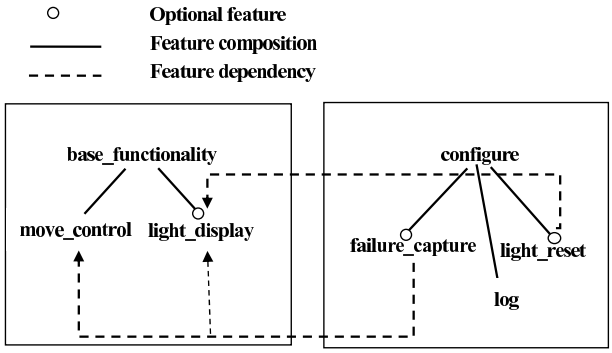

Fig. 3: Simplified example of feature models for an elevator system

a mandatory feature for *base_functionality* and an optional feature *configure* for customised configuration. Moreover, the *base_functionality* is a product family that includes a mandatory feature *move_control* and an optional feature *light_display*. For the *configure* feature, we consider two unpredictable variabilities, *light_reset* and *failure_capture*. Inherently, the *light_reset* depends on the *light_display* and the *failure_capture* depends on both the features *move_control* and *light_display*. Besides, we also consider a mandatory feature *log* that is included into the feature *configure* due to an evolution process.

Figure 3 illustrates feature models using a FODA-like notation[1] for the *base_functionality* and *configure*, respectively. Assume Specification 1 in Figure 4 is the initial PFA specification of the elevator product family. In this specification, Lines 1–3 specify three basic features and Lines 4–9 specify product families as labelled product family algebra terms. Line 10 is a constraint, which indicates that, within the product family *elevator_product_line*, the feature *configure* requires the feature *light_display*.

[1] FODA (Kang et al, 1990) is one of the feature-modeling techniques that are widely used to specify the reusable core assets of a product family and to describe how individual products can be configured from these core assets.

Specification 1:

```
    % Declarations of basic features
1.  bf move_control
2.  bf light_display
3.  bf configure
    % Definitions of labeled product families
4.  optional_light_display = light_display+ 1 % an optional feature
5.  optional_configure = configure + 1
6.  full_base_functionality = move_control · light_display
7.  base_functionality = move_control · optional_light_display
8.  elevator_product_line = base_functionality · optional_configure
9.  customised_elevators = move_control + full_base_functionality
                           · configure
    % Articulating a constraint on the family
10. constraint(configure, elevator_product_line, light_display)
```

Fig. 4: A PFA Specification of the Elevator Product Family

In Section 4, we illustrate how to integrate features *light_display*, *failure_capture*, and *log* to the original specification in various ways using the proposed aspect-oriented technique.

3 Aspect Orientation at the Feature Level

We extend the aspect-oriented notions to PFA specifications of feature models. We call the proposed language *AO-PFA (Aspect-Oriented Product Family Algebra)*. In product family algebra, all kinds of common and variable characteristics of product families are described and unified as product family terms. In other words, the basic constructs of product family algebra specifications are product family terms. Intuitively, join-points in our technique should be in the form of product family terms and the point-cut language defines quantification statements over those product family terms. Based on the mathematical setting of PFA specifications, an aspect in AO-PFA is compactly specified as follows:

$$\begin{array}{ll} \text{Aspect} & \langle aspectId \rangle = \langle Advice(jp) \rangle \\ where & jp \in \big(scope,\ expression,\ kind\big) \end{array}$$

The triple $(scope, expression, kind)$ is the point-cut language of AO-PFA, which specifies the quantification statement for selecting join-points. The equation $\langle aspectId \rangle = \langle Advice(jp) \rangle$ is the body of the aspect which specifies the advice being introduced at selected join-points. The grammar of the language for aspect specifications is given in Figure 5, where ϵ denotes the empty string. In the remainder of this section, we present a detailed discussion on join-points, point-cuts, advice, and aspects in AO-PFA.

⟨AspectSpec⟩ := (⟨Aspect⟩\n)$^+$
⟨Aspect⟩ := ⟨aspectId⟩ = ⟨Advice(jp)⟩\n where jp ∈ ⟨POINTCUT⟩
⟨aspectId⟩ := *identifiers of aspects*
⟨Advice(jp)⟩ := *product family terms defined in PFA using a variable 'jp'*
⟨POINTCUT⟩ := (*base*, ⟨EXPRESSION_BASED⟩, ⟨Constraint-related⟩)
|(⟨SCOPE⟩, ⟨EXPRESSION_BASED⟩, ⟨Feature-related⟩)
| (⟨SCOPE⟩, ⟨EXPRESSION_BASED⟩, ⟨Family-related⟩)
⟨SCOPE⟩:=⟨SCOPE⟩ ; ⟨SCOPE⟩|⟨SCOPE⟩ : ⟨SCOPE⟩|*base*
| *within*{⟨PF_label⟩}| *hierarchy*{⟨PF_label⟩}|*protect*{⟨PF_label⟩}
⟨EXPRESSION_BASED⟩:=*Boolean expression upon PFA*
⟨Feature-related⟩:=*declaration*{⟨PFT⟩}|*inclusion*{⟨PFT⟩}
⟨Family-related⟩:=*creation*{⟨PFT⟩}|*component_creation*{⟨PFT⟩}
|*component*{⟨PFT⟩}|*equivalent_component*{⟨PFT⟩}
⟨Constraint-related⟩:=*constraint*[⟨*list*⟩]{⟨PFT⟩}
⟨list⟩:=*left*⟨list'⟩|*middle*⟨list'⟩|*right*⟨list'⟩
⟨list'⟩:=, *left*⟨list'⟩|, *middle*⟨list'⟩|, *right*⟨list'⟩|ϵ
⟨PFT⟩:=*product family terms defined in PFA.*
⟨PF_label⟩:=*identifiers of product families.*

Fig. 5: Language for Aspect Specifications

3.1 Join-Points in AO-PFA

We have mentioned above that join-points in PFA specifications are in the form of product family terms. However, within a PFA specification, there are two roles for the same form of product family terms. They are either being defined or being referenced. For example, in Figure 4, the family *base_functionality* is being defined at the left side in Line 6, while it is being referenced at the right side in Line 8. Consequently, there are two types of join-points: *definition join-points* and *reference join-points*. Integrating new aspects at the two types of join-points corresponds to two different situations when handling the requirements. Roughly speaking, the specified product family term can be considered as a white box in the former case, whereas it can be considered as a black box in the latter case. Introducing advice at a definition join-point affects the internal description of the specified product family term, whereas introducing advice at a reference join-point affects the descriptions of product families including the specified product family terms. Moreover, when it comes to the detailed level of features, introducing advice at these two types of positions can cause very different results. Therefore, it is necessary to distinguish between the definition and reference positions of a product family term at the abstract feature-modeling level. The differences between these two types of join-points are discussed further when specifying point-cuts, advice, and aspects.

3.2 Point-cuts in AO-PFA

In existing aspect-oriented techniques, three attributes are generally used to specify a point-cut: the scope of join-points, a predicate that characterises the relevant join-points, and the form and role of join-points. Therefore, the point-cut language in AO-PFA is expressed as a triple (*scope*, *expression*, *kind*).

The first component of the point-cut triple, *scope*, bounds the selecting scope of join-points in PFA. Two types of scopes are designed: *within* and *hierarchy*. Scopes of type *within* capture join-points within specified lexical structures, while scopes of type *hierarchy* capture join-points within the hierarchical property of features in the feature models. We use ":" and ";" to express the combination of two scopes. Separating two scopes by ":" indicates that all eligible join-points are within the union of the two specified scopes. Separating two scopes by ";" indicates that all eligible join-points are within the intersection of the two specified scopes. Moreover, we use *protect* to specify that all eligible join-points are excluded from the scope. In particular, when no scope is specified, the scope *base* is considered by default, indicating that the whole base specification is in the scope.

The second component of the point-cut triple, *expression*, is a Boolean expression on the language of product family algebra, which captures characteristics of the product families corresponding to the base specification. Boolean expressions work as guards for the selected join-points. When no expression point-cut is specified, the expression *true* is taken by default.

The third component of the point-cut triple, *kind*, is used to specify the exact form and role of join-points. Unlike the scopes and the expressions of point-cuts, there is no default value for the kind of a point-cut. The kind of point-cut must be explicitly specified for each aspect. With regard to the three types of specification constructs in PFA, we further categorise the kinds of point-cuts as feature-related (*declaration* and *inclusion*), family-related (*component_creation*, *component*, *creation*, and *equivalent_component*), and constraint-related (*constraint*[*position_list*]). We further illustrate the usage of different kinds of point-cuts in the following sections.

3.2.1 Feature-related Point-cuts

In PFA specifications, feature-related point-cuts capture join-points which are product family terms that are basic features in product families. Two kinds of feature-related point-cuts are introduced. In particular, *declaration* point-cuts capture join-points where a specific feature is declared, while *inclusion* point-cuts capture join-points where a specific feature is referenced. The difference between these two kinds of point-cuts

Table 1: Summary of Types of Point-cuts

Scope	
Default	*base*
Explicit Scope	*hierarchy*
	within
Excluded Scope	*protect*(⟨scope⟩)
Combined Scope	$\langle \text{scope}_1 : \text{scope}_2 \rangle$
	$\langle \text{scope}_1 ; \text{scope}_2 \rangle$

Expression	
Default	true
Explicit Expression	Boolean Expression

Kind		
	Definition	**Reference**
Feature-related	*declaration*	*inclusion*
Family-related	*creation*	*component*
	component_creation	*equivalent_component*
Constraint-related		*constraint*[*position_list*]

resides in whether or not the feature's definition can be changed.

3.2.2 Family-related Point-cuts

Family-related point-cuts capture join-points that are product family terms representing product families in a PFA specification. We introduce four kinds of family-related point-cuts: *creation*, *component_creation*, *component*, and *equivalent_component*.

Creation point-cuts and *component_creation* point-cuts capture join-points at the left sides of labelled family equations, which indicate the definition of specified product families. The difference between *creation* point-cuts and *component_creation* point-cuts resides in whether we change the definition of the specified families directly or whether we change the definition of their components. *Creation* point-cuts refer to the exact definition of the specified families.

On the other hand, *component* point-cuts and *equivalent_component* point-cuts capture join-points at the right sides of labelled family equations, which indicate the reference to the specified product families. *Component* point-cuts refer to the appearance of the specified product families within any other product families as components. *Equivalent_component* point-cuts refer to the equivalent (or indirect) appearance of the specified product families as components. The difference between *component* point-cuts and *equivalent_component* point-cuts resides in whether the reference is direct or indirect.

3.2.3 Constraint-related Point-cut

With regard to constraints in PFA specifications, we introduce a constraint-related point-cut. As each constraint item is represented by a triple preceded by the keyword *constraint*, an extra option in the point-cut is necessary to specify which arguments of the triple are captured. Therefore, the constraint-related point-cut is expressed as *constraint*[*position_list*]. Three keywords, *left*, *middle*, and *right*, respectively correspond to the first, second and third arguments of a PFA constraint triple. The keywords are used to specify the *position_list*.

Table 1 summarises the various types of each elements of the triple (*scope*, *expression*, *kind*).

3.3 Advice and Aspects in AO-PFA

The advice of an aspect is specified by an equation in AO-PFA: $\langle aspectId \rangle = \langle Advice(jp) \rangle$. An aspect can either relate to definition join-points or to reference join-points. With regard to the different types of join-points, there is a slight difference for specifying $\langle aspectId \rangle$. If the aspects relate to definition join-points, $\langle aspectId \rangle$ should specify new labels that define new product family terms. If the aspects relate to reference join-points, $\langle aspectId \rangle$ should always be expressed as a variable *jp* that refers to join-points.

Additionally, we discuss and categorise aspects according to the effects of their advice on the selected join-points, which indicates that the form of the advice in AO-PFA is always specified by a product family term; either a ground term or a term with variable *jp*. In particular, we distinguish aspects in accordance with their augmenting, narrowing, and replacing effects upon join-points.

3.3.1 Augmentation Aspects

Augmentation aspects add features to the original specifications. In other words, for an augmentation aspect, the advice is specified by a product family term constructed with variable *jp*. We further classify augmentation aspects with respect to definition join-points and reference join-points. *Refine* aspects augment the original product families where they are defined, whereas *extend* aspects augment original product families where they are referenced.

Table 2: The categories of Aspects

Type of Join-Points	Effects on Join-Points	Categories
Definition Join-Points	Augmentation	*refine*
	Narrowing	*discard*
	Replacement	*replace*
Reference Join-Points	Augmentation	*extend*
	Narrowing	*disable*
	Replacement	*substitute*

3.3.2 Narrowing Aspects

Narrowing aspects simply result in the absence of original join-points. The advice of narrowing aspects can be specified as the constant element 1 of product family algebra. This means that a product or family is replaced by the neutral product denoted by 1 (a pseudo-product that has no features). Similar to augmentation aspects, narrowing aspects are further classified into *discard* and *disable* aspects. *Discard* aspects narrow product families or basic features where they are defined, whereas *disable* aspects narrow product families or basic features where they are referenced.

3.3.3 Replacement Aspects

Replacement aspects replace the appearance of original join-points with other product families. In this case, the advice can be specified in the form of a ground product family term (i.e., a term constructed without variables). Similarly, we distinguish *replace* aspects and *substitute* aspects to respectively refer to effects on definition join-points and reference join-points.

As mentioned earlier, the type of join-points is decided by the kind of the point-cut. In particular, *declaration*, *creation*, and *component_creation* capture definition join-points, while *inclusion*, *component*, *equivalent_component*, and *constraint*[*position_list*] capture reference join-points. Moreover, the effects of aspects on join-points are decided by the form of $\langle Advice(jp)\rangle$. Therefore, given the syntax of an aspect in AO-PFA, we can directly categorise the aspect. Such a classification of aspects is to help the modular reasoning on aspects in the context of product families. In summary, the categories of aspects are given in Table 2.

4 Specifying Aspects with AO-PFA

After a product family has been deployed, it is common that the product families evolve. Families of ambient systems are good examples of product families where evolution is unavoidable and continuous (Gámez and Fuentes, 2012). Different circumstances require either the addition of new features according to new techniques or requirements, or the replacement of certain features by other ones, or simply the removal of some bad or old features from the product families. The proposed language has the flexibility to implement each of these different types of changes by composing aspects. In particular, the augmentation, replacement, and narrowing aspects respectively correspond to adding, replacing, and removing features. Besides adding, replacing, and removing features, different evolution scenarios have distinct requirements that should be able to be specified by using aspects. In the remainder of this section, we further discuss the example of the elevator product family to illustrate the flexibility of the proposed language. Figures 6 – 8 show the resulting specifications after weaving different aspects. In those specifications, we use bold font to denote join-points in the base specification and italic font to denote new specification elements introduced by the aspect.

4.1 Introduction of Similar Crosscutting Concerns Generating Different Sets of Products

A common type of requirement for product families is that the appearance of certain features implies the appearance of other features. The deployed product families should be able to evolve with such requirements. Meanwhile, the way to evolve with such requirements can vary according to different circumstances.

Case (a): Suppose that we wish to express a new requirement that intends to compose a new *light_reset* feature in any family that includes a *light_display* feature. Take Specification 1 as the base specification. We can use an aspect with an *inclusion* point-cut to specify this scenario.

Aspect	$jp = jp \cdot$ *light_reset*
where	$jp \in$ (base, true, *inclusion*(light_display))

For this aspect, the new feature *light_reset* is composed with *light_display* where *light_display* is referenced in the labelled product families. In particular, the captured join-points in Specification 1 are at the right hand sides of both Line 4 and Line 6. The resulting specification is Specification 2 of Figure 6. Moreover, according to the classification described in Section 3.3, this is an *extend* aspect.

Case (b): Suppose that we wish to express a new requirement that intends to introduce two optional features, *failure_capture* and *light_reset*, to the original definition of *configure*. Take Specification 1 of the elevator

product family given in Figure 4 as our base specification. We can use an aspect with a *declaration* point-cut to specify this scenario as follows:

Aspect	*jp_new = (1 + failure_capture) · (1 + light_reset)*
where	*jp* ∈ (base, true, *declaration*(configure))

The point-cut here would capture a join-point related to the definition of *configure* at Line 3 of Specification 1. Consequently, since the captured and modified join-point is a definition join-point and the scope point-cut is *base*, all references to the original *configure* should be substituted to the new ones. The resulting specification is Specification 3 of Figure 6. This aspect is a *replace* aspect.

Case (c): We next take Specification 3 of Figure 6 as the base specification. Since the original *configure* feature has been changed to *configure_new*, the constraints related to *configure_new* that are inherited from the original constraints related to *configure* may become too restrictive or too loose. In particular, the constraint at Line 12 in Specification 3 is automatically generated from the constraint at Line 10 in Specification 1 by substituting *configure* by *configure_new*. However, this constraint cannot exactly specify the relationship between the *light_display* and the newly added feature *light_reset*. Therefore, we specify an aspect with a *constraint*[*position_list*] point-cut as follows:

Aspect	*jp = light_reset*
where	*jp* ∈ (base, true, *constraint*[*left*](configure_new))

The above aspect is a *substitute* aspect. The kind of point-cut captures join-points which reference the feature *configure_new* and appear at the left components of a constraint triple. Obviously, the captured join-points are the first component of Line 12 in Specification 3 of Figure 6. The result of weaving this aspect to Specification 3 is given in Specification 4 of Figure 6.

We can find that the aspects in Case (b) and Case (c) together specify similar requirements to the aspect in Case (a). In both situations, a new *light_reset* feature is introduced in the elevator product family and the appearance of *light_reset* indicates the appearance of *light_display* in each product. In Case (a), as shown in Specification 2, the new *light_reset* feature is only composed with products where *light_display* is available in the base Specification 1. On the other hand, the aspect in Case (b) first introduces the new *light_reset* feature by further defining *configure* in the base Specification 1. Consequently, weaving the aspect in Case (c) to the base Specification 3 indicates that *light_reset* requires *light_display* in all products of the elevator product family. However, we should notice that the product families specified by the above two situations are not exactly the same. The former one composes a new feature to the original product families to make them all satisfy the requirement (see Specification 2 of Figure 6), while the latter one removes the products that do not satisfy the requirements from the product family (see Specification 4 of Figure 6).

Specification 2: Using an *inclusion* point-cut

```
...
    bf light_reset
...
4.  optional_light_display = light_display · light_reset +1
...
6.  full_base_functionality  = move_control · light_display ·light_reset
...
```

Specification 3: Using a *declaration* point-cut

```
1.  bf move_control
2.  bf light_display
3.  bf failure_capture
4.  bf light_reset
5.  configure_new = (failure_capture+1) · (light_reset +1)
6.  optional_light_display = light_display + 1
7.  optional_configure = configure_new + 1
8.  full_base_functionality = move_control · light_display
9.  base_functionality = move_control · optional_light_display
10. elevator_product_line = base_functionality · optional_configure
11. customised_elevators = move_control + full_base_functionality
                           · configure_new
12. constraint(configure_new, elevator_product_line, light_display)
```

Specification 4: Using a *constraint*[*position-list*] point-cut

```
...
12. constraint(light_reset, elevator_product_line, light_display)
```

Fig. 6: Different Cases of Adding the New Feature *light_reset*

4.2 Introduction of Similar Crosscutting Concerns at Different Types of Join-Point Positions

To add new features, it is usually the case that we need to compose a new feature with a particular product family. Since a feature can have different roles at different places in the base specification, we need to add those new features at different types of join-points.

Case (a): Consider Specification 3 of Figure 6 to be the base specification. Suppose that we want to include a new *log* feature with the original *configure_new* product family. We can accomplish this using a *refine* aspect as follows:

Aspect	*jp_new = jp · log*
where	*jp* ∈ (base, true, *creation*(configure_new))

The *creation* point-cut intends to capture join-points where *configure_new* is defined. Therefore, this aspect captures join-points at the left hand side of Line 5 of Specification 3 and adds a new feature *log* to obtain a new definition for *configure_new*. Since the captured and changed join-points are definition join-points,

Specification 5: Using a *creation* point-cut

```
    bf log
...
5.  configure_new = (failure_capture + 1) · (light_reset + 1)
    configure_new_new = configure_new · log
...
7.  optional_configure = configure_new_new + 1
...
11. customised_elevators = move_control + full_base_functionality
                         · configure_new_new
12. constraint(configure_new_new, elevator_product_line, light_display)
```

Specification 6: Using a *component* point-cut

```
    bf log
...
7.  optional_configure = configure_new · log + 1
...
11. customised_elevators = move_control + full_base_functionality
                         · configure_new · log
...
```

Fig. 7: Different Cases of Adding the New Feature *log*

Specification 7: Using a *component_creation* point-cut

```
1.  bf move_control
2.  bf light_display
    bf failure_capture
    move_control_new = move_control · failure_capture
    light_display_new = light_display · failure_capture
...
6.  full_base_functionality = move_control_new · light_display_new
...
```

Specification 8: Using an *equivalent_component* point-cut

```
...
    bf failure_capture
...
6.  full_base_functionality = move_control · light_display
7.  base_functionality = move_control · light_display · failure_capture
                       + move_control
...
9.  customised_elevators = move_control +full_base_functionality ·
                         failure_capture · configure
...
```

Specification 9: Using a non-default scope point-cut

```
    bf failure_capture
...
9.  customised_elevators = move_control + move_control · failure_capture
                         · light_display · configure
...
```

Fig. 8: Different Cases of Adding the New Feature *failure_capture*

the references to *configure_new* in Line 7, Line 10, and Line 12 are automatically substituted by the new one. Specification 5 of Figure 7 shows the result of weaving this aspect to Specification 3.

Case (b): We assume that a new feature *log* is required to be composed with the *configure_new* product family as described above, while the original definition of *configure_new* should not be changed. In this case, we can use an *extend* aspect to specify this scenario as follows:

Aspect	$jp = jp \cdot log$
where	$jp \in (\text{base, true, } component(\text{configure\_new}))$

The *component* point-cut intends to capture join-points where *configure_new* is referenced in the labelled product families. Therefore, this aspect composes the feature *log* at the right hand sides of Line 7 and Line 10 of Specification 3 where *configure_new* appears. The result of weaving this aspect to Specification 3 is shown in Specification 6 of Figure 7.

Comparing the resulting specifications of the above two cases, the difference lies in whether or not the definitions of product families are changed. The aspect in Case (a) modifies the original definition of *configure_new*. All references to the feature *configure_new* in the specification, including the one in the constraint, have to be changed to the new one in Specification 5. On the other hand, the aspect in Case (b) does not change the original definition of *configure_new* and the reference to this product family in the constraint is unchanged in Specification 6.

4.3 Introduction of Similar Crosscutting Concerns Regarding Different Feature Relationships

We still consider adding new features to the original product family. However, the newly added feature may not only have simple relationships with one particular feature/family in the original product family as in the previous section. We should be able to add new features in more flexible ways that can handle more complex relationships.

Case (a): Suppose that we want to capture any defective behaviour in the *full_base_functionality*. However, *full_base_functionality* is composite and we cannot be sure which component might cause the defective behaviour. Therefore, we need to add a *failure_capture* feature to each of its components, *move_control* and *light_display*. Take Specification 1 (Figure 4) as our base specification. We can specify an aspect with a *component_creation* point-cut as follows:

Aspect	$jp\_new = jp \cdot failure\_capture$
where	$jp \in (\text{base, true, } component\_creation(\text{full\_base\_functionality}))$

The components of *full_base_functionality*, which is specified in Line 6, are *move_control* and *light_display*. Consequently, the join-points related to the definitions of *move_control* and *light_display* are Line 1 and Line 2. In other words, the above point-cut would capture join-points at both Line 1 and Line 2 of Specification 1 of Figure 4, and add the new feature *failure_capture* there. Automatically, references to those components *move_control* and *light_display* in Line 6 are changed

to the new ones. Specification 7 of Figure 8 shows the result of weaving this aspect to Specification 1. The aspect is a *refine* aspect.

Case (b): Alternatively, suppose we want to capture any equivalent defective behaviour in the product family *full_base_functionality* from the base specification. However, assume that we are not allowed to make changes to the definition of the product family *full_base_functionality*. An aspect with an *equivalent_component* point-cut, exemplified in Section 3.2, is able to specify this scenario. Taking Specification 1 of Figure 4 as our base specification again, we specify an aspect with an *equivalent_component* point-cut as follows:

Aspect	$jp = jp \cdot$ *failure_capture*
where	$jp \in$ (base, true, *equivalent_component*(full_base_functionality))

For this aspect, the captured join-points should be reference join-points which are equivalent to the product family *full_base_functionality*. The right hand side of Line 7 includes the product family term *move_control* · *light_display*, which is equivalent to the product family term *full_base_functionality* due to the definition of *full_base_functionality* at Line 6. Moreover, the right hand side of Line 9 includes the product family term *full_base_functionality*. Therefore, this aspect will add the new feature *failure_capture* at the right hands sides of Line 7 and Line 9 of Specification 1. Specification 8 of Figure 8 shows the result of weaving this aspect to Specification 1. Straightforwardly, the aspect is an *extend* aspect according to the proposed classification.

Case (c): We continue with our running example to introduce a new *failure_capture* feature in the base specification. Suppose we are required to capture all defective behaviours with the *move_control* component in the *full_base_functionality* family. In addition, we only introduce the new feature within the *customised_elevators*. In this case, we specify an aspect as follows:

Aspect	$jp = jp \cdot$ *failure_capture*
where	$jp \in$ (*within*(customised_elevators) ; *hierarchy* (full_base_functionality), true, *inclusion*(move_control))

Straightforwardly, the captured join-points should be related to the reference of *move_control*. Moreover, the set of join-points are further bounded by the scope point-cuts. In particular, the *within* scope narrows the join-points to only Line 9 of Specification 1 of Figure 4, and the *hierarchy* scope specifies that the *move_control* should be constructed from the *full_base_functionality*. Therefore, this aspect will not add the *failure_capture* with the first *move_control*, but with the second one in Line 9. This is an *extend* aspect and the resulting specification is Specification 9 of Figure 8.

Although the term of the advice is the same for each aspect (i.e., *Advice*(*jp*) is $jp \cdot$ *failure_capture* for each of the above cases), the resulting specifications are quite different. The join-points of the aspects in Case (a) and Case (b) are related to *full_base_functionality*. The join-points of the aspect in Case (c) are related to the feature *move_control*, while the feature *full_base_functionality* only specifies the scope of join-points. Furthermore, besides the slight difference in meaning, the main difference between Case (a) and Case (b) resides in whether or not the definitions of the *full_base_functionality* family (or its components) have changed. The different effects of these aspects show that our point-cut language is capable of distinguishing between slight differences among requirements.

5 Remarks on AO-PFA

5.1 Theoretical Remarks

The language of a product family algebra is that of a semiring. However, the extended PFA language includes equations defining families and constraints on families presented in Section 2.1, which brings some complications into its semantics. A product family specification is a parameterised algebraic specification (SPEC, PSPEC). The component SPEC is the parameter specification, and the component PSPEC is the target specification. Then, a particular product family algebra specification can be denoted by the actualisation of the parameter SPEC. The parameter specification SPEC contains a sort f, and operations $+$, $\cdot$, 0 and 1 of the commutative idempotent semiring. The sort f denotes the sort of product families. The parameter specification is a subspecification of the target specification, which means that the target specification can use the sorts, and the operations defined in the parameter specification. We have a set E_f of equations which are the axioms of a commutative idempotent semiring. An equation is usually represented by a triple (X, l, r) which means $l = r$ with $l, r \in T_f(X)$. For example, an equation $(x+y)+z = x+(y+z)$ is represented by $\big(\{x, y, z\}, (x+y)+z, x+(y+z)\big)$, where $\{x, y, z\}$ explicitly claims the variables that occur in the equation. Formally, the parameterised specification is then defined as follows:

$$\text{PSPEC}[\text{SPEC}] = \text{SPEC} \uplus (\emptyset, \emptyset, E_f), \tag{1}$$

where $\uplus$ is the disjoint union.

A given PFA specification S can be denoted as a term algebraic specification with a fixed set of symbols $\mathfrak{L}(S)$, which represent the labels of the defined families (including basic features). Product family terms

w.r.t. the specification S are denoted by $T_f(\mathfrak{L}(S))$. The signature of S is $(\{f\},\ \{+,\cdot,1,0\}\cup\mathfrak{L}(S)\})$. Moreover, let $(\{c\},*)$ be the signature of a monoid. The set of all possible labels for features or families can be denoted by $V = T_c(\Gamma)$, where Γ is the alphabet and $*$ is interpreted as the concatenation operation on strings. We should have $\mathfrak{L}(S) \subseteq V$. Besides, a PFA specification S also brings a set of equations $Eq(S)$ defining families, and a set of constraints $Cp(S)$. Each equation (X, l, r) in $Eq(S)$ satisfies $l, r \in T_f(X)$ and $X \subseteq \mathfrak{L}(S)$. Each constraint in $Cp(S)$ is represented by a triple (l, m, r) satisfying $l, r \in T_f(\mathfrak{L}(S))$, corresponding to the requirement relationship introduced in Section 2.1. Therefore, a given PFA specification S corresponds to an algebraic specification

$$\mathcal{S} = \big(\{f\},\ \{+,\cdot,1,0\}\cup\mathfrak{L}(S),\ Eq(S)\cup Cp(S)\big).$$

According to Equation (1), we denote the given PFA specification S by actualising the parameter specification SPEC with $\mathcal{S}$, which can be written as

$$\text{PSPEC}[\text{SPEC}\mapsto\mathcal{S}] = \mathcal{S}\uplus(\emptyset,\emptyset,E_f).$$

In summary, the semantics of the PFA specification language are those of $\mathcal{S}\uplus(\emptyset,\emptyset,E_f)$, which are still semiring-based semantics. The models can be trivial extensions to models of commutative idempotent semirings such as the set-based or the bag-based ones discussed in (Höfner et al, 2006). In AO-PFA, we have a base specification that is a PFA specification and an aspect specification. After the weaving process, we get a PFA specification (for more details, we refer the reader to (Zhang and Khedri, 2013)). We have shown in (Zhang and Khedri, 2013) that the rewriting system needed for the weaving process is convergent (i.e., confluent and terminating). Also, we have shown that the procedure for determining a join-point in the base specification is decidable (i.e., deciding on $E_f \cup Eq(S) \models (s = t)$, for terms s and t).

5.2 Applicability of AO-PFA

To evaluate the applicability and the effectiveness of the proposed approach, we discuss four of the desired qualities that should be facilitated by any analysis and design approach: composability, evolvability, scalability, and traceability (Chitchyan et al, 2005).

The composition of feature models with AO-PFA is rigorously specified by the point-cut language, and is well-defined according to the formal semantics of the PFA language and the weaving process (Zhang and Khedri, 2013). Moreover, the formal techniques given in (Zhang et al, 2011, 2012b) provide techniques for verifying the correctness of aspectual composition.

The proposed feature-modeling approach is evolvable due to the characteristics of the aspect-oriented paradigm. The case study given in Section 4 illustrates the flexibility of the proposed approach for making changes to feature models. The support of the formal weaving progress and the formal verification of the aspectual composition problems make amending feature models automatic, which eases the evolution of the models. Any change to the feature model can be systematically and automatically propagated.

The scalability of the proposed approach is constrained by the scalability of the *Jory* tool that is used to specify and analyse PFA specifications. *Jory*, which executes PFA specifications for analysis purposes, uses the binary decision diagrams provided by the library BuDDy (Lind-Nielsen, 2010). BuDDY can handle up to $50,000$ nodes in every megabyte of memory (tested for 2^{32} nodes) (Alturki and Khedri, 2010). We use one node to represent one feature. Therefore, the handled families can be huge in size (Alturki and Khedri, 2010).

The traceability of the proposed approach is not addressed at this point in time. It requires techniques that map features from the feature-modeling level to the design and implementation levels. As future work we intend to develop mapping techniques from an AO-PFA specification to the elements of all the following stages of the development life-cycle.

6 Related Work and Discussion

6.1 Work Related to Product Family Algebra

The work of AO-PFA is based on the work of product family algebra (Höfner et al, 2006) and a formal specification language PFA (Alturki and Khedri, 2010). Formal specifications are particularly essential for being used by tools to assist complex activities such as managing and analysing large and complicated feature models. Product family algebra is notable for its capability to represent feature models very compactly. Moreover, the *Jory* tool implements the automatic analysis of PFA specifications based on BDDs (Binary Decision Diagrams) (Andersen, 1997). Results shown in (Benavides et al, 2006), indicate that BDDs provide a faster approach for the analysis and reasoning of feature models. Therefore, we can particularly benefit from using this tool for extremely large feature models.

Our work is also an extension of the work in (Höfner et al, 2008). In this work, feature models are integrated by view integration, where each view partially describes common and variable characteristics of the considered product family. View reconciliation (Höfner et al, 2008) is used to exclude all products violating the constraints

from the integrated product families. However, as recognised in general software development, there is a bottleneck for the classic multi-view integration approach to separate and compose crosscutting concerns (Chitchyan et al, 2005). If we compose crosscutting concerns with other concerns by view reconciliation, we are required to specify various constraints in the corresponding specification. To specify all of those constraints can be quite tedious for large feature models. Moreover, adding and removing features from feature models are common operations that are necessary for the evolution of feature models. But constraints only allow to remove products from the original product families. The language proposed in this paper intends to complement the multi-view integration approach used in product family algebra by providing a modular means to handle crosscutting concerns. A complete product family algebra specification for a product family feature model is described by integrating different views/perspectives of the system with view reconciliation and composing crosscutting concerns with the aspect-oriented paradigm. Similar to the general benefits that can be obtained from the aspect-oriented paradigm, our technique can improve the modular development, maintenance, and evolution of feature models for product families.

Besides, there are also other extension works for AO-PFA, such as the verification of aspectual composition in AO-PFA. To avoid obtaining unsound feature models by composing aspects with base specifications, we have proposed a formal verification approach for aspectual composition in (Zhang et al, 2012b). In particular, the work of (Zhang et al, 2012b) establishes a set of criteria and propositions to enable the detection of dependency, reference, or definition invalid aspects before the weaving process based on AO-PFA. The verification of aspectual composition is a common and heavily discussed issue in the literature of aspect-oriented techniques (Kuhlemann et al, 2009). However, the main difficulty for aspectual verification in many early aspect-oriented techniques is due to their informal specifications. Therefore, the formal nature of AO-PFA eases the formal verification of aspectual composition at the early stages. Moreover, the formal verification of aspectual composition in AO-PFA also provides a way to detect unsafe composition of features at a more abstract level and at an earlier stage.

6.2 Approaches for Using Product Family Engineering for Ambient Systems

In (Fuentes and Gámez, 2010), we find a discussion of why and how to use product family engineering in ambient systems. Ambient systems are comprised of a set of heterogeneous devices, which communicate via diverse environments. Besides, different applications of ambient systems have quite distinct requirements. Conventionally, one solution to handle the complexity and variabilities of ambient systems is to use a middleware platform. However, with such a middleware solution, all applications deployed in each device will be homogeneous. In other words, more services are deployed but are disabled in each one of those heterogeneous devices. On the other hand, product family engineering can provide a rapid and systematic way to develop similar but specific ambient systems that target heterogeneous devices, different networks, or distinct requirements.

Moreover, the evolution problems in ambient systems require approaches to deal with them in the context of product family engineering. We find in (Gámez and Fuentes, 2012) a process to automatically propagate and trace changes from feature models to architecture components of ambient systems. Since their work does not discuss the evolution problems of the feature models themselves, our work can be considered as a complement work of theirs, which provides a systematic way for changing the feature models.

6.3 Approaches for Adapting the Aspect-Oriented Paradigm into Product Family Engineering

Our work mainly adapts several ideas from AspectJ to the product family specification level. Terms in our work can be analogous to terms in AspectJ. On the other hand, by constructing our language upon the mathematical structure of product family algebra, we simplify the constructs and notations which would be used in aspect-oriented techniques. Besides modularisation, the aspect-oriented paradigm also provides the mechanism for composition. In contrast with AspectJ, feature-related point-cuts can be respectively analogised as the field set and field get point-cuts in AspectJ. The two types of family-definition point-cuts can be connected to the class/object creation point-cuts in AspectJ, while the other two types of family-reference point-cuts can be connected to the method-related point-cuts in AspectJ. However, unlike at the programming level, we are able to adapt aspect-oriented techniques at the abstract level of product families using very simple notations with the help of product family algebra. In PFA, only a few notations are necessary to specify product families. Hence, join-points of PFA specifications can simply be unified as product family terms, which are the basic elements in PFA specifications.

We find other related works that attempt to manage the variabilities in product families by introducing the aspect-oriented paradigm to product family en-

gineering. Griss (Griss, 2000) summarised the advantages of composition and weaving approaches for dealing with crosscutting concerns in product families. Earlier techniques that adapt the aspect-oriented paradigm to product family engineering mainly focus on the implementation stage at the programming level, such as Framed aspect (Loughran and Rashid, 2004), Aspectual Mixin Layer (Apel et al, 2006), and Caesar (Mezini and Ostermann, 2004). Recently, several works are articulated around adapting the aspect-oriented paradigm at the earlier analysis and design stages. VML4RE (Alférez et al, 2009) presents a requirement specification language to compose elements from different requirement models. The language is based on concrete models that it supports (i.e., use cases, interaction diagrams, and goal models). Xweave (Groher and Voelter, 2007) is a model weaver supporting the composition of different views. It helps weaving variable parts of architectural models to base models. In VML4RE and Xweave, the variabilities of product families are composed into the concrete models of product families using the aspect-oriented paradigm. Their techniques can be seen as complement techniques of our method. By appropriately mapping mechanisms for aspects, we should handle aspects consistently and systematically from the feature-modeling level to the concrete models and to the implementation.

In (Bošković et al, 2010), the authors introduce a feature-modeling technique AoFM and argue that the aspect-oriented paradigm can help to manage and maintain large feature models. This work has a similar motivation as ours. However, due to the graph-based notations of AoFM, it is a challenge to automate the processing and verification of feature models. In comparison with AoFM, our technique adopts a formal aspect-oriented approach at the feature-modeling level, which enhances automatic analysis and verification of feature models.

6.4 Approaches for Composing Feature Models

At the modeling and specification level for product families, many efforts have been taken in the literature to manage the common and variable features. Our work aims to facilitate the management of complexity in large feature models. In (Acher et al, 2010), the authors deal with a problem similar to that of our work, but in a different way. With regard to the composition of feature models, they mainly focus on the insert and merge operators. From our perspective, their merge operator can be handled by using view reconciliation presented in (Höfner et al, 2008, 2009) and the insert operator can be handled with the aspect-oriented paradigm. Their work considers the composition operators from the perspective of model integration, whereas our work discusses the issue from the perspective of composition mechanisms for different concerns.

Another focus of our approach is formalisation. A feature algebraic foundation for feature composition is proposed in (Apel et al, 2010). Similar to product family algebra, the work captures the basic ideas of features and feature composition at an abstract level in terms of an algebraic setting. However, their approach for feature composition is more related to programming languages. Our approach resides in the specification language at the early analysis and design stages of product family engineering. This enables us to bridge the gap of formalisations from the requirements stage to the implementation stage. Another related work is discussed in (Thüm et al, 2009) that deals with reasoning on feature models. In their work, they explore the relationship between the original feature model and the modified feature model. By contrast, our technique handles feature models modularly with the aspect-oriented paradigm and explores the relationship between base feature models and composed feature models by reasoning on aspects. In this paper, we classify aspects to help with modular reasoning on them.

7 Conclusion and Future Work

In this paper, we presented an approach which adapts the aspect-oriented paradigm to feature-modeling techniques of product families. We proposed a language, AO-PFA, which extends aspect-oriented notations to specifications based on product family algebra. The proposed language allows for the articulation of aspects, advice, and point-cuts in feature-modeling. The semantics of the proposed language is based on the models of product family algebra discussed in (Höfner et al, 2006, 2009). Also, through the use of several feature-modeling situations and an illustrative example of an elevator product family, we illustrated the scope and flexibility of the proposed language.

Ambient systems are able to adapt to the particular needs and profiles of a variety of users. They also offer its users mobile and pervasive access to data. For this reason, ambient systems are exceedingly susceptible to change. As the needs of the users and the environment change, developers of ambient systems need to be able to amend the system in a timely manner in order for the system to cope with such changes. Although the environments for which ambient systems are developed contain many common characteristics, they are not altogether the same. Similarly, different users may have different needs, but in all likelihood, they will share a

number of common needs. Due to this commonality, the recommended approach for developing these types of ambient systems is the family oriented approach. In this paper, we proposed an approach for specifying ambient systems as product families. By proposing the use of an aspect-oriented approach, we provide a convenient means for amending the feature models of ambient systems to allow them to evolve in order to fit their environments and meet the needs of their users.

We intend to extend the existing *Jory* tool to automate the proposed approach. The envisioned extension, would use the kernel of *Jory* for the automatic analysis of large feature models. With regard to the syntax of the proposed language, we only need to perform a lightweight extension to the existing notation of *Jory* to support the aspect specifications. In addition, we aim at developing modules to support the verification of aspectual composition and the weaving of aspects. Based on the formal technique presented in (Zhang et al, 2011), the verification of aspectual composition for the proposed approach can be easily automated. Also, we have given the formal semantics and proofs for the weaving process in (Zhang and Khedri, 2013), according to which the automation of the weaving process is straightforward. In particular, with regard to the term rewriting system required by the weaving process, we have implemented a prototype using the term rewriting tool *Maude* (Clavel et al, 2011). After completing the above extensions to the tool support of the proposed approach, we aim at conducting an empirical evaluation to assess its performance on industrial-sized feature models.

As the basis for our ongoing and future work on the introduction of finer granularity aspects at the state level rather than the feature level, we use the work presented in (Höfner et al, 2011) to analyse feature models at a finer level that of the systems' states. We aim to advance towards automatic code generation from the specification of the base product family, the specification of the aspects, and the specification of each of the basic features. This was shown to be an achievable objective in (Höfner et al, 2011), where the features of a product family were given as requirements scenarios formalised as pairs of relational specifications of a proposed system and its environment.

Acknowledgements This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Grant Number RGPIN227806-09.

We thank the numerous reviewers for their comments that helped us improve the quality of this paper.

References

Acher M, Collet P, Lahire P, France R (2010) Composing feature models. In: van den Brand M, Gaševic D, Gray J (eds) Software Language Engineering, Lecture Notes in Computer Science, vol 5969, Springer Berlin / Heidelberg, pp 62–81

Alférez M, Santos J, Moreira A, Garcia A, Kulesza U, Araújo J, Amaral V (2009) Multi-view composition language for software product line requirements. In: Proc. of the 2nd International Conference on Software Language Engineering, pp 103–122

Alturki F, Khedri R (2010) A tool for formal feature modeling based on bdds and product families algebra. In: 13th Workshop on Requirement Engineering, pp 109–120

Andersen HR (1997) An introduction to binary decision diagrams. In: Lecture notes for 49285 Advanced Algorithm E97

Apel S, Leich T, Saake G (2006) Aspectual mixin layers: Aspects and features in concert. In: Proc. of the International Conference on Software Engineering, pp 122–131

Apel S, Lengauer C, Möller B, Kästner C (2010) An algebraic foundation for automatic feature-based program synthesis. Science of Computer Programming 75(2010):1022–1047

Benavides D, Segura S, Trinidad P, Ruiz-Cortes A (2006) A first step towards a framework for the automated analysis of feature models. In: Proc. of the Workshop held in conjunction with the 10th Software Product Line Conference

Bošković M, Mussbacher G, Bagheri E, Amyot D, an Marek Hatala DG (2010) Aspect-oriented feature models. In: Models in Software Engineering-Workshops and Symposia at MODELS 2010

Chitchyan R, Rashid A, Sawyer P, Garcia A, Alarcon MP, Bakker J, Tekinerdogan B, Clarke S, Jackson A (2005) Survey of Analysis and Design Approaches. Survey, AOSD-Europe

Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2011) Maude Manual (Version 2.6). SRI International, Menlo Park, CA 94025, USA.

Czarnecki K (1998) Generative programming, principles and techniques of software engineering based on automated configuration and fragment-based component models. PhD thesis, Technical University of Ilmenau

Eriksson M, Börstler J, Borg K (2005) The PLUSS approach-domain modeling with features, use cases and use realization. In: Proc. of 9th International Conference on Software Product Lines, pp 33–44

Frei R, Di Marzo Serugendo G, Şerbănuţă TF (2010) Ambient intelligence in self-organising assembly systems using the chemical reaction model. Journal of Ambient Intelligence and Humanized Computing 1(3):163–184

Frei R, Şerbănuţă TF, Di Marzo Serugendo G (2012) Self-organising assembly systems formally specified in Maude. Journal of Ambient Intelligence and Humanized Computing pp 1–20

Fuentes L, Gámez N (2010) Configuration process of a software product line for AmI middleware. Journal of Universal Computer Science 16(12):1592–1611

Gámez N, Fuentes L (2012) Architectural evolution of FamiWare using cardinality-based feature models. Information and Software Technology

Griss ML (2000) Implementing product-line features by composing component aspects. In: Proc. of First International Software Product Lines Conference, pp 271–288

Griss ML, Favaro J, d'Alessandro M (1998) Integrating features modeling with the RSEB. In: Proc. of the 5th International Conference on Software Reuse, pp 76–85

Groher I, Voelter M (2007) Xweave: Models and aspects in concert. In: Proc. of the 10th Workshop on Aspect-Oriented Modelling, pp 35–40

Habib S, Marimuthu P (2011) Self-organization in ambient networks through molecular assembly. Journal of Ambient Intelligence and Humanized Computing 2(3):165–173

Höfner P, Khedri R, Möller B (2006) Feature algebra. In: Misra J, Nipknow T, Sekerinski E (eds) Formal Methods, Lecture Notes in Compute Science, vol 4085, Springer-Verlag, pp 300–315

Höfner P, Khedri R, Möller B (2008) Algebraic view reconciliation. In: Proc. of 6th IEEE International Conference on Software Engineering and Formal Methods, pp 85–94

Höfner P, Khedri R, Möller B (2009) An algebra of product families. Software and Systems Modeling 10(2):161–182

Höfner P, Khedri R, Möller B (2011) Supplementing product families with behaviour. International Journal of Informatics pp 245 – 266

Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) Feature oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Kuhlemann M, Batory D, Kästner C (2009) Safe composition of non-monotonic features. In: 8th International Conference on Generative Programming and Component Engineering

Kuusela J, Tuominen H (2009) Aspect-Oriented approach to operating system development empirical study. Journal of Communication and Computer 6(8):233–238

Lind-Nielsen J (2010) Buddy BDD Library. http://sourceforge.net/projects/buddy/, (Last accessed on March 28, 2013)

Loughran N, Rashid A (2004) Framed aspect: Support variability and configurability for AOP. In: Proc. of International Conference on Software Reuse, pp 127–140

Mezini M, Ostermann K (2004) Variability management with feature-oriented programming and aspects. In: Proc. of the 12th ACM International Symposium on Foundations of Software Engineering, pp 127–136

Nygard KE, Xu D, Pikalek J, Lundell M (2010) Multiagent designs for ambient systems. In: 1st International ICST Conference on Ambient Media and Systems, pp 10:1–10:6

Parnas DL (1976) On the design and development of program families. IEEE Transactions on Software Engineering 2(1):1–9

Riebisch M, Böllert K, Streitferdt D, Philippow I (2002) Extending feature diagrams with UML multiplicities

Solhaug B, Seehusen F (2013) Model-driven risk analysis of evolving critical infrastructures. Journal of Ambient Intelligence and Humanized Computing pp 1–18

Thüm T, Batory D, Kästner C (2009) Reasoning about edits to feature models. In: In Proc. International Conference on Software Engineering

Zhang Q, Khedri R (2013) Proofs of the convergence of the rewriting system for the weaving of aspects in the AO-PFA language. Tech. Rep. CAS-13-01-RK, McMaster University, Hamilton, Ontario, Canada, available: `http://www.cas.mcmaster.ca/cas/0template1.php?601`

Zhang Q, Khedri R, Jaskolka J (2011) An aspect-oriented language based on product family algebra: Aspects specification and verification. Tech. Rep. CAS-11-08-RK, McMaster University, Hamilton, Ontario, Canada, available: `http://www.cas.mcmaster.ca/cas/0template1.php?601`

Zhang Q, Khedri R, Jaskolka J (2012a) An aspect-oriented language for product family specification. In: Ambient Systems, Networks and Technologies, 3rd International Conference, ANT2012

Zhang Q, Khedri R, Jaskolka J (2012b) Verification and validation of aspectual composition in product families. In: Software Engineering and Formal Methods, 10th International Conference, SEFM2012