Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 10 (2012) 482 – 489

Procedia Computer Science

The 3rd International Conference on Ambient Systems, Networks and Technologies (ANT)

An Aspect-Oriented Language for Product Family Specification☆

Qinglei Zhang, Ridha Khedri, Jason Jaskolka

Department of Computing and Software, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada

Abstract

Aspect-orientation is a paradigm for managing the separation of crosscutting concerns and decomposing a system using more than one criterion. This paper proposes an aspect-oriented approach at the feature-modeling level to better handle crosscutting concerns in the modeling of product families of ambient systems.

Based on the specification language of PFA (Product Family Algebra), we present a language AO-PFA (Aspect-Oriented Product Family Algebra) that extends the aspect-oriented paradigm to feature modeling. The language provides full facilities for articulating aspects, advice, and pointcuts in feature modeling. We illustrate the scope and flexibility of the proposed language through the discussion of several feature-modeling situations.

Keywords: software product families, aspect-oriented software development, early aspects, formal methods, formal specification languages

1. Introduction and Motivation

Ambient systems involve a multitude of heterogeneous features that are interconnected and that supply end users with a variety of data and functionality. Their stability is contingent on requirements that can cope with high variability. Due to the amount of data collected from the environment and the complexity of the hardware and software involved in collecting and acting on it, ambient systems' environments pose special challenges to the feature-modeling process. The variety of hardware leads to a variety of possible technologies, which in turn, leads to some predictable variability of the family of similar systems. This point has been pointed to by Parnas [1] as early as 1976. Product family modeling was proposed to deal with this problem. It proposes the simultaneous development of a family of products, rather than of one product at a time. Concerning the variability in the requirements of systems in general, there are two classes of variability: predictable and unpredictable. There are some requirements that predictably vary and therefore can be included in the feature model of the product family from the beginning. For instance, if we were modeling a family of robots, with our knowledge of the available technology, we can predict that we might have different collision detection systems (e.g. infra-red based and radio based). We then would

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

Email addresses: zhangq33@mcmaster.ca (Qinglei Zhang), khedri@mcmaster.ca (Ridha Khedri), jaskolj@mcmaster.ca (Jason Jaskolka)

1877-0509 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2012.06.062

have systems with different detection features that we include from the start in the feature model of a robot product family. One situation that illustrates unpredictability of some features of a family of systems can be easily illustrated with a security related situation. To remotely communicate with an ambient system, we often use an authentication feature that is in charge of identifying the caller agent. After a while, we find that there is a flaw in the authentication feature due to the presence of other features. In other words, the feature interaction of the authentication feature and other features make it possible for an intruder to take control of some systems of the family. The question then becomes how to quickly amend the current feature model to ensure that all the product families that involve the identified configuration of features gets amended to replace the flawed configuration by another configuration of features. This change to the family due to the security issue cannot be predicted at the time of the feature modeling of the family. When the flaw is revealed, we might have several systems that are already deployed in their environment. Sometimes, the remedy of the detected defect in a product family leads to the introduction of new variability in the family or to the amendment of the existing variability by confining it to some products but not others.

Dealing with predictable variability is the topic of a wide literature of feature-modeling techniques [2, 3, 4, 5, 6]. Unpredictable variability, however, is somewhat limited. In dealing with ambient systems, one cannot dismiss unpredictable variability and it ought to be given a wide attention. A flexible approach to adapt the feature models regarding unpredictable variabilities in various ways are demanded. Especially, the dynamic characteristic of ambient systems make the quick change to the system a priority that ought to be carried out in a speedy manner. Consequently, it is critical to develop an adaptable and evolvable systems while managing the complexity of the systems. Modularization of concerns is essential to manage complexity of ambient systems. However, some concerns are inherently spread over and intertwined with other concerns, and therefore resist such modularization by conventional approaches. As illustrated in [7], multi-agent systems, which is an example of ambient systems, are associated with many crosscutting concerns such as autonomy, communication, mobility, security, etc.. As hindering the maintainability and modifiability of software qualities, crosscutting concerns make the ambient systems difficult to be adapted and evolved. Aspect-oriented software development addresses the modularization of crosscutting concerns, and provides a powerful way to handle crosscutting concerns in ambient systems. We find that software engineering deals with a similar problem at the programming level using aspect-oriented techniques. However, at the programming level, these techniques showed very mixed results. Aspect-oriented programming leads to systems with high modifiability, but at the same time the performance is hindered [8]. Also, 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. These problems are very minimal at the feature-modeling level. That is why we conjecture that aspect-oriented techniques, while they exhibited mixed results at the programming level, can be helpful at the feature-modeling level.

This paper is about a language which is built on the language of product family algebra that would enable us to systematically amend a product family to deal with unpredictable changes as soon as they are revealed. We would have the specification of the family that is to be amended, which we call the base specification, and add to it the specification of the aspect we ought to address. It is through the weaving process that we generate the specification of the amended family. The base specification of a family is commonly referred to as the feature model. This paper builds on the work presented in [9, 10, 11, 12] by expanding the language of PFA to an aspect-oriented language.

The remainder of the paper is organised as follows. Section 2 introduces the related background knowledge. Section 3 presents the proposed specification language and its usage. Section 4 discusses other works reported in the literature of aspect-oriented software development and product family engineering. Finally, in Section 5, we conclude and give 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 operator $\cdot$. An element $0 \in S$ is the identity element with respect to +,

⟨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.

(a) PFA Specification Grammar

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.  base_functionality = move_control · light_display
7.  optional_base_functionality =  move_control
        · optional_light_display
8.  full_functional_elevator = base_functionality · configure
9.  elevator_product_line = optional_base_functionality
        · optional_configure
    % a constraint
10. constraint(configure, elevator_product_line, light_display)
```

(b) Base Specification of The Elevator Product Family

Fig. 1. PFA Language Grammar and an Example

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 operator $\cdot$ is commutative and a semiring is idempotent if the operator $+$ is idempotent.

Definition 1 (e.g., [11]). *A product family algebra is a commutative idempotent semiring* $(S, +, \cdot, 0, 1)$*, where each element of the semiring is a product family.*

Within the context of product family engineering, the operator $+$ is interpreted as a choice between two product families and the operator $\cdot$ is interpreted as a mandatory composition of two product families. The element 0 represents the empty product family and the element 1 represents a product family consisting of only a pseudo-product which has no features. Optional features can be interpreted as a choice between the features and the pseudo-product 1. With these interpretations, all other concepts in product family modeling can be expressed mathematically. Formal definitions for features, products, and families can be found in [9, 11].

For elements a and b of 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 of 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 of a product family algebra, the requirement relation ($\rightarrow$) is defined in a family-induction style as: $a \xrightarrow{p} b \iff_{df} p \sqsubseteq a \Longrightarrow p \sqsubseteq b$, and $a \xrightarrow{c+d} b \iff_{df} a \xrightarrow{c} b \wedge a \xrightarrow{d} b$. Given elements a, b, and c, $a \xrightarrow{c} b$ is read as "a requires b within c".

A tool based on product family algebra, called Jory [13], 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(a). In PFA, there are three types of specification constructs: basic feature declarations, labeled 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 families 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. Constraints are specified for view reconciliation [10] of product families. Specification 1 in Figure 1(b) specifies an elevator product family using the PFA language of Jory. In this specification, Lines 1–3 specify three basic features, Lines 4–9 specify product families as labeled terms based on product family algebra, and Line 10 specifies a constraint as defined in product family algebra. According to product family algebra, for example, an optional feature *light_display* is specified by a term *light_display* +1 in Line 4. In Line 6, the *base_functionality* is specified by the term *move_control* · *light_display*, meaning the family *base_functionality* includes the two mandatory features *move_control* and *light_display*. Line 10 corresponds

to *configure* $\overset{elevator\_product\_line}{\longrightarrow}$ *light_display* in product family algebra, meaning that in the product family *elevator_product_line*, the feature *configure* requires the feature *light_display*. The reader can find more details on the use of this mathematical framework to specify product families in [9, 11, 12].

We will use the elevator system as a running example in this paper. We consider two unpredictable variabilities, *light_reset* and *failure_capture*, in the product family. Inherently, the *light_reset* feature depends on the *light_display* feature, and the *failure_capture* depends on both the *move_control* and the *light_display*. Moreover, we consider a mandatory feature *log* is added to the feature model of *configure* due to an evolution process. Assume Specification 1 in Figure 1(b) is the initial PFA specification of the elevator product family. In Section 3, we illustrate how to integrate features *light_display*, *failure_capture*, and *log* to the original specification using the proposed aspect-oriented technique.

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. 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 *pointcut* 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, pointcuts identify join points where an aspect should be introduced, while advice defines the specification of the crosscutting concern.

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 pointcut 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:

Aspect	$\langle aspectId \rangle = \langle Advice(jp) \rangle$
where	$jp \in (scope, expression, kind)$

The triple $(scope, expression, kind)$ is the pointcut 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 new advice being introduced at selected join points. In the remainder of this section, we present a detailed discussion on join points, pointcuts, advice, and aspects in AO-PFA.

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 1(b), the product 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 an advice at a definition join point affects the internal description of the specified product family term, whereas introducing an 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 pointcuts, advice, and aspects.

3.2. Pointcuts in AO-PFA

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

The first component of the pointcut 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 eligible join points are within the union of the two specified scopes. Separating two scopes by ";" indicates that eligible join points are within the intersection of the two specified scopes. Moreover, we use *protect(scope)* to specify that 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 pointcut 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 pointcut is specified, the expression *true* is taken by default.

The third component of the pointcut triple, *kind*, is used to specify the exact form and position of join points. Unlike scopes and expressions of pointcuts, there is no default value for the kind of a pointcut. The kind of pointcut must be explicitly specified for each aspect. With regard to the three types of specification constructs in PFA, the kinds of pointcuts are specified as feature-related (i.e., *declaration* and *inclusion*), family-related (i.e., *creation*, *component_creation*, and *equivalent_component*), and constraint-related (i.e., *constraint*[*position_list*]). Particularly, *declaration*, *creation*, and *component_creation* pointcuts introduce new specifications at definition join points, whereas *inclusion*, *component*, *equivalent_component*, and *constraint*[*position_list*] pointcuts introduce new specifications at reference join points. Moreover, the difference between *creation* and *component_creation* pointcuts resides in whether we change the definition of the specified families directly or whether we change the definition of their components. The difference between *component* and *equivalent_component* pointcuts resides in whether the reference is direct or indirect.

3.3. Advice and Aspects in AO-PFA

As given previously, the body of an aspect is specified by an equation $\langle aspectId \rangle = \langle Advice(jp) \rangle$. According to the effect of an aspect upon join points (i.e., augmenting, narrowing, and replacing), we indicate that $\langle Advice(jp) \rangle$ is always specified by a product family term; either a ground term or a term with variable *jp*.

Augmenting aspects add features to the original specifications. In other words, for an augmenting aspect, the advice is specified by a product family term constructed with variable *jp*. We further classify augmenting 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.

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 augmenting 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.

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.

With regard to $\langle aspectId \rangle$, there is a slight difference for specifying aspects that relate to different types of join points. 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. Furthermore, as mentioned earlier, the type of join

⟨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.*

(a) Aspect Specification Grammar

Specification 2: Using *component_creation* pointcut

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. **base_functionality** = **move_control_new** · **light_display_new**

Specification 3: Using *equivalent_component* pointcut

...
... bf *failure_capture*
6. base_functionality = move_control · light_display
7. option_base_functionality = **move_control** · **light_display**
   · *failure_capture* + move_control
8. full_functional_elevator = **base_functionality** · *failure_capture*
   · configure
...

Specification 4: Using non-default scope pointcut

... bf *failure_capture*
9. elevator_product_line = move_control + **base_functionality**
   · *failure_capture* + **full_functional_elevator**
   · *failure_capture*
...

(b) Resulting PFA specifications

Fig. 2. Weaving Aspects

points is decided by the kind of pointcut. Therefore, given the syntax of an aspect in AO-PFA, we can directly categorise the aspect according to its form of ⟨*Advice*(*jp*)⟩ and the third component of the pointcut triple. Such a classification of aspects is to help the modular reasoning on aspects in the context of product families.

3.4. Specifying Pointcuts, Advice and Aspects with AO-PFA

Taking Specification 1 of Figure 1(b) as the base specification, we use several examples to illustrate the usage and flexibility of the proposed language. The grammar of the proposed language for aspect specifications is given in Figure 2(a). More examples detailing the usage of all constructs can be found in [14].

(1) Suppose that we want to capture any defective behaviour in the product family *base_functionality*. However, the *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*. To specify this requirement, we use an aspect with a *component_creation* pointcut as follows:

Aspect	*jp_new* = *jp* · *failure_capture*
where	*jp* ∈ (base, true, *component_creation*(base_functionality))

The *component_creation* pointcut refers to the definitions of all components in the *base_functionality*. The resulting specification is Specification 2 of Figure 2(b). According to the classification described in Section 3.3, the aspect is a *refine* aspect.

(2) Alternatively, suppose that we want to capture any equivalent defective behaviour in the *base_functionality* product family from the base specification. However, assume that we are not allowed to make changes to the definition of the product family *base_functionality*. This requirement is able to be specified by an aspect with an *equivalent_component* pointcut as follows:

Aspect	*jp* = *jp* · *failure_capture*
where	*jp* ∈ (base, true, *equivalent_component*(base_functionality))

The *equivalent_component* pointcut refers to all equivalent appearances, (i.e., both direct and indirect references) of the *base_functionality*. The resulting specification is Specification 3 of Figure 2(b). Straightforwardly, the aspect is an *extend* aspect according to the proposed classification.

(3) 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 *base_functionality* family. In addition, we only introduce the new feature within the *elevator_product_line* family. The aspect below with a non-base scope pointcut can be used to express this requirement.

Aspect	$jp = jp \cdot failure\_capture$
where	$jp \in (within(\text{elevator\_product\_line}) ; hierarchy(\text{base\_functionality}), \text{true}, inclusion(\text{move\_control}))$

The *inclusion* pointcut captures join points where the feature *move_control* is referenced. The *within* scope of the pointcut narrows the scope of join points to only Line 9 of Specification 1. Since *hierarchy* specifies that the feature *move_control* should be constructed from the family *base_functionality*, we do not compose *failure_capture* with the first *move_control* in Line 9. Specification 4 of Figure 2(b) shows the result of weaving this aspect to Specification 1. The above aspect is also an *extend* aspect.

Although the term of the advice is the same for each aspect (i.e., $Advice(jp)$ is $jp \cdot failure\_capture$), the resulting specifications are quite different. The join points of the aspects in Case (1) and Case (2) are related to *base_functionality*. The join points of the aspect in Case (3) are related to *move_control*, while *base_functionality* only specifies the scope of join points. Furthermore, besides the slight difference in meaning, the main difference between Case (1) and Case (2) resides in whether or not the definitions of the *base_functionality* family (or its components) have changed. The different effects of these aspects show that our pointcut language is capable of distinguishing between slight differences among requirements.

4. Related Work and Discussion

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. Acher et al. [15] 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 [10, 11], 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. We find other related works [16, 17, 18, 19, 20, 21] that attempt to manage the variabilities in product families by introducing the aspect-oriented paradigm to product families. In comparison with those approaches, our technique introduces the aspect-oriented paradigm at the feature-modeling level. By appropriately mapping mechanisms for aspects, we should handle aspects consistently and systematically from the feature-modeling level to the concrete models and the implementation. Therefore, their techniques can be seen as complement techniques to our method.

5. Conclusion and Future Work

In this paper, we introduced the aspect-oriented paradigm to feature-modeling techniques of product families. We presented AO-PFA which extends aspect-oriented notations to specifications based on product family algebra. The proposed language provides full facilities for articulating aspects, advice, and pointcuts in feature modeling. The semantics of the language is based on the models of product family algebra that are discussed in [9, 11]. We illustrated the scope and flexibility of the proposed language through the discussion of several feature-modeling situations. It is important to ensure that composing new aspects will not invalidate the original specification of base systems. Since our approach is constructed upon a formal setting, it is easier for us to formally verify the validities of aspects with regard to base systems. In [14], we have already established a set of criteria and propositions for formally detecting invalid aspects before weaving them to base systems.

Ambient systems are systems that offer its users mobile and pervasive access to data and that are able to adapt themselves to the particular user needs and profiles. Therefore, they ought to be systems that are very prone to change. As users needs and the environment change, the developers need to quickly amend the systems to cope with these changes. The environment for which these systems are developed are not the same, however, they share common characteristics. Also, the users might have different needs but very likely have common shared needs. Therefore, a family oriented approach is the approach recommended for developing these systems. The paper proposes an approach to specify ambient systems as product families. To enable ambient systems to evolve in order to fit their environments and their users needs, the paper proposes the use of an aspect-oriented approach to amend their feature models.

We are using the work presented in [12] as the basis for our ongoing work on introducing finer granularity aspects at the state level rather than at the feature level. The objective of this work is to get closer to automatic code generation from the specification of the product family base, the specification of the aspects, and the specification of the basic features. Höfner et al. [12] showed that it is an achievable objective. They present the features of a family as requirements scenarios formalised as pairs of relational specifications of a proposed system and its environment. The result of weaving aspects should lead to, among others, the specification of a product presented using a slight variation of Dijkstra's guarded command [22].

References

[1] D. L. Parnas, On the design and development of program families, IEEE Trans. Software Eng. 2 (1) (1976) 1–9.
[2] K. Czarnecki, Generative programming, principles and techniques of software engineering based on automated configuration and fragment-based component models, Ph.D. thesis, Technical University of Ilmenau (Oct. 1998).
[3] M. Eriksson, J. Börstler, K. Borg, The PLUSS approach-domain modeling with features, use cases and use realization, in: Proc. of 9th International Conference on Software Product Lines, 2005, pp. 33–44.
[4] M. L. Griss, J. Favaro, M. d'Alessandro, Integrating features modeling with the RSEB, in: Proc. of the 5th International Conference on Software Reuse, 1998, pp. 76–85.
[5] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (Nov 1990).
[6] M. Riebisch, K. Böllert, D. Streitferdt, I. Philippow, Extending feature diagrams with UML multiplicites (2002).
[7] K. E. Nygard, D. Xu, J. Pikalek, M. Lundell, Multi-agent designs for ambient systems, in: 1st International ICST Conference on Ambient Media and Systems, 2010, pp. 10:1–10:6.
[8] J. Kuusela, H. Tuominen, Aspect-Oriented Approach to Operating System Development Empirical Study, Journal of Communication and Computer 6 (8) (2009) 233–238.
[9] P. Höfner, R. Khedri, B. Möller, Feature algebra, in: J. Misra, T. Nipknow, E. Sekerinski (Eds.), Formal Methods, Lecture Notes in Computer Science, Vol. 4085, Springer-Verlag, 2006, pp. 300–315.
[10] P. Höfner, R. Khedri, B. Möller, Algebraic view reconciliation, in: Proc. of 6th IEEE International Conference on Software Engineering and Formal Methods, 2008, pp. 85–94.
[11] P. Höfner, R. Khedri, B. Möller, An algebra of product families, Software and Systems Modeling 10 (2) (2011) 161–182.
[12] P. Höfner, R. Khedri, B. Möller, Supplementing product families with behaviour, International Journal of Informatics (2011) 245 – 266.
[13] F. Alturki, R. Khedri, A tool for formal feature modeling based on bdds and product families algebra, in: 13th Workshop on Requirement Engineering, 2010, pp. 109–120.
[14] Q. Zhang, R. Khedri, J. Jaskolka, 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` (Nov. 2011).
[15] M. Acher, P. Collet, P. Lahire, R. France, Composing feature models, in: M. van den Brand, D. Gaševic, J. Gray (Eds.), Software Language Engineering, Vol. 5969 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 62–81.
[16] M. Alférez, J. Santos, A. Moreira, A. Garcia, U. Kulesza, J. Araújo, V. Amaral, Multi-view composition language for software product line requirements, in: Proc. of the 2nd International Conference on Software Language Engineering, 2009, pp. 103–122.
[17] S. Apel, T. Leich, G. Saake, Aspectual mixin layers: Aspects and features in concert, in: Proc. of the International Conference on Software Engineering, 2006, pp. 122–131.
[18] M. L. Griss, Implementing product-line features by composing component aspects, in: Proc. of First International Software Product Lines Conference, 2000, pp. 271–288.
[19] I. Groher, M. Voelter, Xweave: Models and aspects in concert, in: Proc. of the 10th Workshop on Aspect-Oriented Modelling, 2007, pp. 35–40.
[20] N. Loughran, A. Rashid, Framed aspect: Support variability and configurability for AOP, in: Proc. of International Conference on Software Reuse, 2004, pp. 127–140.
[21] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, in: Proc. of the 12th ACM International Symposium on Foundations of Software Engineering, 2004, pp. 127–136.
[22] E. Dijkstra, A discipline of programming, Prentice-Hall, 1976.