An Aspect-Oriented Language Based on Product Family Algebra: Aspects Specification and Verification

by

Qinglei Zhang, Ridha Khedri, and Jason Jaskolka

C.A.S. **CAS-11-08-RK**

Department of Computing and Software **November 2011**
Information Technology Building
McMaster University
1280 Main Street West Hamilton, Ontario, Canada L8S 4K1

An Aspect-Oriented Language Based on Product Family Algebra: Aspects Specification and Verification*

Qinglei Zhang, Ridha Khedri, and Jason Jaskolka†

Technical Report CAS-11-08-RK

Department of Computing and Software

McMaster University

November 11, 2011

Abstract

Aspect-orientation is a promising 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 product families. The implementation of some features of a product family can be inherently scattered over several features or tangled within other features. The development of such features bears the same problems as general crosscutting concerns. The Aspect-oriented paradigm provides an effective means for handling crosscutting concerns.
Based on the language of Product Family Algebra (PFA), 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, advices, and pointcuts in feature modeling. Moreover, we present a formal verification technique of aspectual composition in the context of AO-PFA. We define a set of validity criteria for aspects with regard to their corresponding base specifications. The proposed approach enables the detection of dependency, reference, or definition invalid aspects.

Keywords: Software product families; Feature-modeling; Early aspects; Aspect-oriented paradigm; Formal methods; Formal specification languages; Requirements verification

Contents

[†]This research is supported by Natural Sciences and Engineering Research Council of Canada (NSERC) through the grant RGPIN227806-09.

[†]E-mail: {zhangq33, khedri, jaskolj}@mcmaster.ca

1 Introduction

Product Family Engineering is an emerging software development approach which aims to improve productivity, increase quality, and decrease cost, labour and time to market [40]. In this approach, new products are derived from core assets rather than developed independently. The commonality and variability among a product family are captured in terms of features. Mapping features of a product family to the design and implementation is not always straightforward. Particularly, there are common and variable features related to crosscutting concerns that inherently scatter over and intertwine with several other features. Consequently, crosscutting concerns impede the extensibility and maintainability of systems in general and product families in particular. Without appropriate modularisation, it is difficult to modularly view and reason about the crosscutting concerns. Furthermore, it is necessary to trace crosscutting concerns from the early analysis stage to the following design and implementation stages. Identifying and treating crosscutting concerns from the early feature-modeling stage is the first step to ease the traceability of crosscutting concerns within the life-cycle of product families.

The aspect-oriented paradigm is a promising technique for managing the separation of crosscutting concerns and decomposing a system using more than one criterion. Roughly speaking, the aspect-oriented paradigm encapsulates crosscutting concerns with aspects and provides an efficient composition mechanism to integrate aspects with base systems. Earlier techniques that adapt the aspect-oriented paradigm to product family engineering mainly focus on the implementation stage at the programming level (e.g., [4, 27, 30]). Recently, several works are articulated around adapting the aspect-oriented paradigm systematically throughout the whole life-cycle of product family engineering (e.g., [29]). Similar to adapting aspect-oriented techniques at the early analysis and design stage for general software development, adapting aspect-oriented techniques at the feature-modeling level provides an effective way to discover and manage aspects early in the product family engineering life-cycle. It is important to keep in mind that for early aspects, it is not necessary that they are mapped as aspects at the implementation level. Early aspects help us to make the early trade-off and provide criteria for design decisions used in the following development stages.

Our research focuses on adapting the aspect-oriented paradigm to a feature-modeling technique called product family algebra [15, 16, 17]. While numerous feature-modeling techniques in the literature are graph-based (e.g., [9, 11, 13, 20, 33]), product family algebra is a formal technique that specifies product families precisely and compactly. Product family algebra enables the algebraic specification of product families. With the extension of mathematical capabilities, product family algebra provides us a ground for further analysis and the ability to manipulate feature models in a calculative way.

In Section 2, we introduce the related background knowledge of our method. In Section 3, we present our specification language that is used to express pointcuts, advices and aspects at the feature level. Moreover, we propose a classification mechanism for aspects within the context of product family engineering. In Section 4.1, we establish the criteria for a valid product family algebra specification and propose tests for detecting invalid aspects in the context of the proposed language. We exemplify the proposed method by a case study of a home automation product line. In Section 5, we conclude our approach and give future work.

2 Background

2.1 Product Family Algebra

Feature-modeling techniques are used to specify and manage the commonality and variability of product families at the domain stage of product family engineering. In the literature, several feature-modeling techniques have been proposed, such as FODA [20], FORM [21], FOPLE [22], FeatuRSEB [13], Generative Programming [9], FORE [38], the Riebisch Technique [33], the van Gurp Technique [39], PLUSS [11], etc..

In [15, 16, 17], Höfner et al. proposed a formal technique, *product family algebra*, to capture a set of different notations and terms found in current feature-modeling techniques. In [3], we find a thorough discussion on how other feature-modeling techniques can be easily translated into specifications in the language of product family algebra. Product family algebra extends the mathematical notations of idempotent 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 $+$, 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 ([17]). *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. In product family algebra, optional features are interpreted as a choice between the features and the pseudo-product 1, which has no features. With these interpretations, all other concepts in product family engineering can be expressed mathematically.

Moreover, constraints are defined in product family algebra for view reconciliation [16]. Definitions 2, 3, and 4 formally give the related concepts.

Definition 2 ([17]). *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.

Definition 3 ([17]). *For elements a and b in a product family algebra, the refinement relation ($\sqsubseteq$) is defined as*

$$a \sqsubseteq b \iff_{df} \exists(c \;|:\; a \leq b \cdot c).$$

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 a has at least all the features of some products in b.

Definition 4 ([17]). *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:*

$$\begin{aligned} a \overset{p}{\rightarrow} b &\iff_{df} p \sqsubseteq a \Longrightarrow p \sqsubseteq b \\ a \overset{c+d}{\rightarrow} b &\iff_{df} a \overset{c}{\rightarrow} b \wedge a \overset{d}{\rightarrow} b \end{aligned}$$

The requirement relation is used to specify constraints in product families. For elements a, b and c, $a \overset{c}{\rightarrow} b$ can be read as "a requires b within c". The reader can find more details on the use of this mathematical framework to specify product families in [15, 16, 17, 18].

Jory [2, 3] is a tool to represent and manipulate product families. The specification language used by Jory is based on product family algebra and called PFA. In PFA, there are three types of syntactic elements: basic feature declarations, labeled product families and constraints. A *basic feature label* preceded by the keyword *bf* declares a basic feature. An equation with a *product families label* at the left-hand side and a product family algebra term at the right-hand side gives a labeled product family. A triple preceded by the keyword *constraint* represents a constraint, which specifies a requirement relation as introduced in Definition 4. The complete grammar of PFA is given in Figure 1.

Take the example of a computer product family used in [17]. A computer product family consists of hardware and software. With regard to hardware, computers are built on hard disks, screens, and printers. With regard software, corresponding drivers for each type of hardware component should be provided. Therefore, in a PFA specification, basic feature declarations include:

bf hard_disk	bf hd_drv
bf screen	bf scr_drv
bf printer	bf prn_drv

In the computer product family, a basic computer is built on a hard disk and a screen, whereas a printer may be added as required. Assume only two different software packages are offered. The product families of hardware and software are specified by labeled product family algebra terms as below:

$$\begin{aligned} \text{hw} &= \text{hard\_disk} \cdot \text{screen} \cdot (1 \;+\; \text{printer}) \\ \text{sw} &= \text{hd\_drv} \cdot \text{scr\_drv} \;+\; \text{hd\_drv} \cdot \text{scr\_drv} \cdot \text{prn\_drv} \end{aligned}$$

$$
\begin{aligned}
&\text{PFA:=}(\langle\text{Basic\_Feature}\rangle \mid \%\langle\text{comment\_txt}\rangle\backslash\text{n})^{+}\\
&\qquad(\langle\text{Labelled\_Family}\rangle \mid \%\langle\text{comment\_txt}\rangle\backslash\text{n})^{+}\\
&\qquad(\langle\text{Constraint}\rangle \mid \%\langle\text{comment\_txt}\rangle\backslash\text{n})^{*}\\
&\langle\text{Basic\_Feature}\rangle\text{:=}\textit{bf}\ \langle\text{base\_feature\_id}\rangle\%\langle\text{comment\_txt}\rangle\backslash\text{n}\\
&\langle\text{Labelled\_Family}\rangle\text{:=}\langle\text{family\_id}\rangle\ =\langle\text{Family\_Term}\rangle\\
&\qquad\qquad\%\langle\text{comment\_txt}\rangle\backslash\text{n}\\
&\langle\text{Constraint}\rangle\text{:=}\textit{constraint(}\langle\text{Family\_Term}\rangle, \langle\text{Family\_Term}\rangle,\\
&\qquad\qquad\langle\text{Family\_Term}\rangle)\%\langle\text{comment\_txt}\rangle\backslash\text{n}\\
&\langle\text{Family\_Term}\rangle\text{:=}0 \mid 1 \mid \langle\text{base\_feature\_id}\rangle \mid \langle\text{family\_id}\rangle\\
&\qquad\qquad\mid \langle\text{Family\_Term}\rangle + \langle\text{Family\_Term}\rangle\\
&\qquad\qquad\mid \langle\text{Family\_Term}\rangle \cdot \langle\text{Family\_Term}\rangle\\
&\langle\text{base\_feature\_id}\rangle\text{:=}\textit{String of letters, numbers and ``\_''}\\
&\langle\text{family\_id}\rangle\text{:=}\textit{String of letters, numbers and ``\_''}\\
&\langle\text{comment\_txt}\rangle\text{:=}\textit{String of letters, numbers , symbols and space}
\end{aligned}
$$

Figure 1: Language of PFA Specifications

Moreover, a requirement that a hard disk requires a hard disk driver is translated in product family algebra into the constraint (hard_disk $\overset{\text{hw} \cdot \text{sw}}{\rightarrow}$ hd_drv). We write this constraint in PFA as follows:

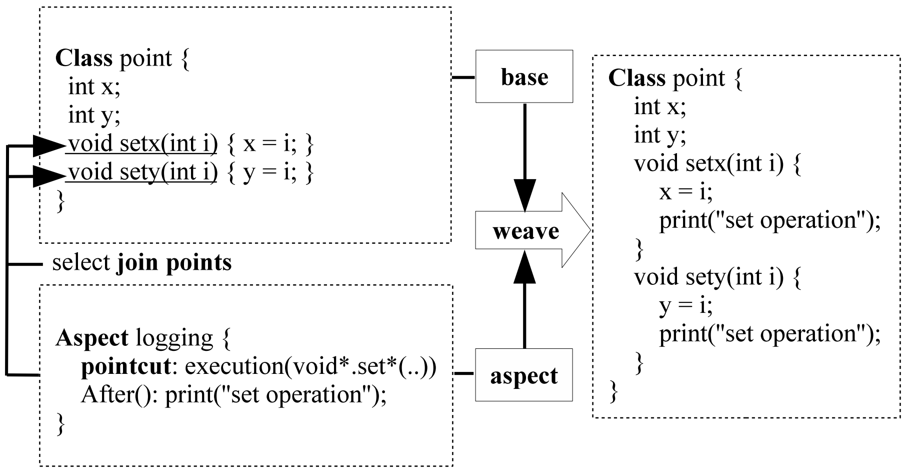
$$\text{constraint(hard\_disk, hw} \cdot \text{sw, hd\_drv)}$$

,

2.2 Aspect-Orientation: Basic Concepts

To better handle crosscutting concerns, the aspect-oriented paradigm encapsulates crosscutting concerns by aspects and provides a particular mechanism for composing aspects with base systems. This paradigm has been adapted to the whole software development life-cycle. According to different granularities of concern abstractions, the meanings of aspects vary at different software development stages. However, several terminologies are widely and commonly used by the community of aspect-oriented software development. First, a *join point* refers to a point at 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 behavior which should be introduced at the selected join points. Lastly, *weaving* is the process of combining aspects with a base program.

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.



```
Class point {
  int x;
  int y;
  void setx(int i) { x = i; }
  void sety(int i) { y = i; }
}
```

```
Aspect logging {
  pointcut: execution(void*.set*(..))
  After(): print("set operation");
}
```

```
Class point {
  int x;
  int y;
  void setx(int i) {
    x = i;
    print("set operation");
  }
  void sety(int i) {
    y = i;
    print("set operation");
  }
}
```

Figure 2: General aspect-orientation mechanism

The pointcut of the logging aspect selects two join points (underlined instructions in the figure) 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 Other Mathematics

We briefly introduce several required graph concepts.

A graph is a 2-tuple $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. For an edge (u, v), u is its tail and v is its head. When $u = v$, we say that (u, v) is a loop. Moreover, u is called the predecessor of v, while v is called the successor of u. We denote the set of all successors of a vertex x as $N^+(x)$ and the set of all predecessors of a vertex x as $N^-(x)$.

A walk in a graph is a list of vertices and edges, such as $v_0, (v_0, v_1), v_1, ..., v_{k-1}, (v_{k-1}, v_k), v_k$. A path is a subgraph which can be represented as a walk without repeated vertices and edges. We use (u, v)–path $\in E^n$ to denote a path starting with vertex u and ending with vertex v, where $n \geq 1$ is the length of the path. In particular, we say a (u, v)–path is a cycle when $u = v$.

3 Aspect Orientation at the Feature Level: AO-PFA

Previous research of product family algebra describes a complete product family feature model by integrating different views/perspectives of the system with view reconciliation. Each view partially describes common and variable characteristics of the considered product family. View reconciliation [16] is used to exclude products that violate constraints after view integration. For example, to describe a computer product family, we integrate feature models that respectively describe the hardware and software views/perspectives.

The view reconciliation is used to guarantee, for instance, that for each valid computer product, its hardware components have appropriate software drivers. However, there are common and variable features related to crosscutting concerns that inherently scatter over and intertwine with several features from several views of the system. As recognized in general software development, there is a bottleneck for the classic multi-view integration approach to separate and compose crosscutting concerns. Our research aims to complement the multi-view integration approach for feature modeling by providing systematic means to handle crosscutting concerns. Particularly, we intend to introduce the aspect-oriented paradigm at the feature-modeling level.

In this section, we extend aspect-oriented notations to product family algebra specifications. We call the proposed language AO-PFA (*Aspect-Oriented Product Family Algebra*). We intend to construct comprehensive specifications of feature models with the proposed aspect-oriented language. At the feature-modeling level, the crosscutting concerns are integrated with the base feature models at the granularity of feature/sub-family abstraction. 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. Moreover, product family terms are also able to express advice, which captures any common and variable characteristics introduced by aspects. Based on the mathematical setting of PFA specifications, an aspect in AO-PFA is compactly specified as follows:

$$\begin{array}{rl} \text{Aspect} & \text{aspectId} = \text{Advice(jp)} \\ \text{where} & \text{jp} \in \Big(\textsf{\textit{Pointcut}}\Big) \end{array}$$

In the the above syntax, *aspectId* represents product family labels that are named by the aspect. The body of the advice is represented as a product family term *Advice(jp)*. The quantification statement for selecting join points is expressed by the pointcut language as given in Figure 3.

In the remainder of this section, we present a detailed discussion on join points, pointcuts, advices, and aspects in AO-PFA.

3.1 Case Study: An Elevator Family

To illustrate the benefits of adapting the aspect-oriented paradigm at the feature-modeling level, we use an example of a simplified elevator system. This elevator example is used as a running example in the remainder of this section. The elevator product family is composed of a feature for *base_functionality* and a *configure* feature for customized configuration. The *base_functionality* includes a mandatory feature *move_control* and an optional feature *light_display*. Included in *configure* is the optional feature *light_reset* and the optional feature *failure_capture*. Inherently, the *light_reset* depends on the *light_display* and the *failure_capture* depends on both the *move_control* and the *light_display*. In other

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⟩}| *cflow*{⟨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⟩:=*identities of product families.*

Figure 3: Pointcut Language

words, the *light_reset* and the *light_display* show crosscutting relations with features in the *base_functionality*. Figure 4 gives the feature model of the elevator product family using FODA-like notations. By adapting the aspect-oriented paradigm, the *light_display* and the *failure_capture* are integrated with the *base_functionality* by composing crosscutting concerns with the base feature model.

Filman and Friedman [12] identify two important characteristics of the aspect-oriented paradigm: *quantification* and *obliviousness*. Obliviousness stipulates that a base system should not be prepared or adjusted to accept newly added or removed crosscutting concerns. Quantification indicates that the composition rules of aspects and the base system are captured by quantification statements. Product family engineering can benefit from the aspect-oriented paradigm in terms of quantification and obliviousness. In product family engineering, it is common to add or delete features with accordance to different configurations. At the feature-modeling level, it is also necessary to support such flexibility of adding or deleting features. With regard to obliviousness, adding or deleting the optional features *failure_capture* and *light_reset* do not have any impact on the *move_control* and the *light_display*. With regard to quantification, when the optional feature *light_display* is added or deleted, the impact is minimized to the compact quantification statement that defines the composition rule for the *light_reset*.

Specification 1 in Figure 5 gives a PFA specification corresponding to the example of the elevator product family. In this specification, Lines 1–3 specify three basic features and Lines 4–9 specify product families by labeled product family algebra terms. Line 10

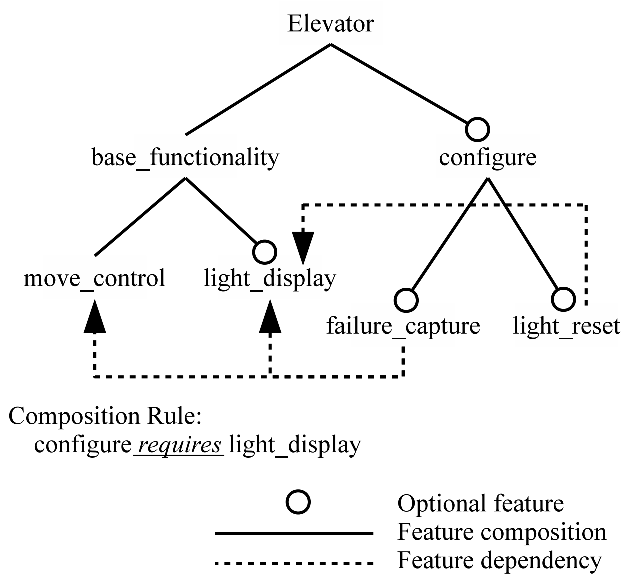


Figure 4: Simplified example of a feature model for an elevator system

is a constraint, which indicates that the *configure* feature requires the *light_display* feature within the product family *elevator_product_line*.

3.2 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 5, the product family *base_functionality* is being defined at the left-hand side in Line 6, while it is being referenced at the right-hand side in Line 8. Consequently, there are two types of join points, the definition join points and the reference join points. Integrating new aspects at the two types of join points corresponds to two different types of requirements scenarios. 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 the definition join points affects the internal description of the specified product family term, whereas introducing an advice at the reference join points affects descriptions of product families including the specified product family term. 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 the definition or reference positions of a product family term at the abstract feature-modeling level. The differences between these two types of join points are further referred for specifying pointcuts, advices, and aspects.

Specification 1: Base elevator product family specification

```
1.   bf move_control
2.   bf light_display
3.   bf configure
4.   optional_light_display = light_display+ 1
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
10.  constraint(configure, elevator_product_line, light_display)
```

Figure 5: PFA specification of the elevator product family

3.3 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 pointcuts. In AO-PFA, we express the pointcut language as a triple (scope, expression, kind).

The first component of the triple corresponds to the *scope* of pointcuts, which bound the selecting scope of join points in PFA. Two types of scope pointcuts are designed: *within* and *cflow*. A scope of type *within* captures join points within specified lexical structures, while a scope of type *cflow* captures join points within a specified construction flow. Here, construction flow refers to 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 pointcut *base* is considered by default indicating that the whole base specification is the scope.

The *expression* component of the triple is designed to capture dynamic properties of join points. It works as guards for the selected join points. To specify the expression component of pointcuts, we use Boolean expressions on the language of product family algebra. When no expression is specified, the expression *true* is taken by default.

The third component of the triple corresponds to the *kind* of pointcuts, which is used to specify the exact form and position of join points. Unlike the scope of pointcuts and the expression of pointcuts, there is no default value for the kind of pointcuts. The kind of a pointcut must be explicitly specified for each aspect. The scope of a pointcut and the expression of pointcut only can take effect when combined with a kind of pointcut. With regard to the three types of specification elements in PFA, we further discuss the kind of pointcuts in our language as feature-related, family-related, and constraint-related pointcuts.

3.3.1 Feature-Related Pointcuts

Two kinds of pointcuts, *declaration* and *inclusion*, are introduced to select basic feature-related join points within PFA specifications. The difference between these two kinds of pointcuts resides in whether or not the feature's definition can be changed.

Declaration pointcuts capture join points where a specific feature is declared. Take Specification 1 of an elevator product family given in Figure 5 as our base specification. To express a new requirement that intends to introduce two optional features to the original definition of *configure*, we specify an aspect as follows:

$$\begin{array}{rl} \text{Aspect} & \text{jp\_new} = (1+\text{failure\_capture}) \cdot (1+ \text{light\_reset}) \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{declaration}(\text{configure})\Big) \end{array}$$

The pointcut here would capture a join point at Line 3 of Specification 1. Automatically, as the scope pointcut is *base*, all references to the original *configure* should be changed to the new one. Figure 6 shows Specification 2, which is the result of weaving this aspect to Specification 1.[1] We use bold font to denote join points in the base specification, and use italic font to denote new specification elements introduced by the aspect.

Inclusion pointcuts capture join points where a specific feature is referenced. Take Specification 1 given in Figure 5 as our base specification. We specify an aspect as below to express a new requirement that intends to compose a new feature *light_reset* in any family including *light_display.*

$$\begin{array}{rl} \text{Aspect} & \text{jp} = \text{jp} \cdot \text{light\_reset} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{inclusion}(\text{light\_display})\Big) \end{array}$$

The pointcut would capture join points at both Line 4 and Line 6 of Specification 1. Figure 7 shows Specification 3 which is the result of weaving this aspect to Specification 1.

3.3.2 Family-Related Pointcuts

We introduce four kinds of family-related pointcuts: *creation*, *component_creation*, *component* and *equivalent_component.*

[1]The old feature *configure* is removed from the specification automatically since there is no reference to it within the whole specification after weaving.

Specification 2:

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. base_functionality = move_control · light_display
9. optional_base_functionality = move_control · optional_light_display
10. full_functional_elevator = base_functionality · **configure_new**
11. elevator_product_line = optional_base_functionality
    · optional_configure
12. constraint(**configure_new**, elevator_product_line, light_display)

Figure 6: Weaving an aspect with a *declaration* pointcut

Specification 3:

. . .

bf *light_reset*

. . .

4. optional_light_display = **light_display** · *light_reset* +1

. . .

6. base_functionality = move_control · **light_display** ·*light_reset*

. . .

Figure 7: Weaving an aspect with an *inclusion* pointcut

Creation pointcuts and *component_creation* pointcuts capture join points at the left-hand sides of labeled families, which indicate the definition of specified product families. The difference between *creation* pointcuts 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.

Creation pointcuts refer to the exact definition of the specified families. We take Specification 2 (Figure 6) as our base specification and specify an aspect as below to extend a new feature *log* in the definition of the product family *configuration_new*.

$$\begin{aligned} &\text{Aspect} && \text{jp\_new} = \text{jp} \cdot \text{log} \\ &\text{where} && \text{jp} \in \Big(\text{base, true, } \textit{creation}(\text{configure\_new})\Big) \end{aligned}$$

The pointcut would capture the left-hand side of Line 5 of Specification 2. Consequently, the references to *configure_new* in Line 7, Line 10, and Line 12 are automatically changed to the new one. Figure 8 shows Specification 4 which is the result of weaving this aspect to Specification 2.

Specification 4:

---

bf *log*

...

5. **configure_new** = (failure_capture + 1) · (light_reset + 1)

configure_new_new = *configure_new* · *log*

...

7. optional_configure = **configure_new_new** + 1

...

10. full_functional_elevator = base_functionality · **configure_new_new**

...

12. constraint(**configure_new_new**, elevator_product_line, light_display)

Figure 8: Weaving an aspect with a *creation* pointcut

Component_creation pointcuts refer to the definitions of all components in the specified families. Take Specification 1 (Figure 5) as our base specification and assume we want to capture any defective behavior in the family *base_functionality*. However, *base_functionality* is composite and we cannot be sure which component might cause the defective behavior. Therefore, we should add a *failure_capture* to each of its components, *move_control* and *light_display*. This requirement can be specified by an aspect as below:

$$\begin{aligned} &\text{Aspect} && \text{jp\_new} = \text{jp} \cdot \text{failure\_capture} \\ &\text{where} && \text{jp} \in \Big(\text{base, true, } \textit{component\_creation}(\text{base\_functionality})\Big) \end{aligned}$$

The pointcut would capture join points at the left-hand sides of both Line 1 and Line 2 of Specification 1 in Figure 5. Automatically, references to those components in Line 6 are changed to the new ones. Figure 9 shows Specification 5 which is the result of weaving this aspect to Specification 1.

Component pointcuts and *equivalent_component* pointcuts capture join points at the right-hand sides of labeled families, which indicate the reference to the specified product families. The difference between *component* pointcuts and *equivalent_component* pointcuts resides in whether the reference is direct or indirect.

Specification 5:

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**
. . .	

Figure 9: Weaving an aspect with a *component_creation* pointcut

Component pointcuts refer to the appearance of the specified product families within any other product families as components. Take Specification 2 (Figure 7) as our base specification. Suppose that a new feature *log* is required wherever product family *configure_new* is included. We specify the aspect as below:

$$\begin{array}{rl} \text{Aspect} & \text{jp}= \text{jp} \cdot \text{log} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{component}(\text{configure\_new})\Big) \end{array}$$

This pointcut would capture join points at Line 7 and Line 10 of Specification 2 in Figure 6. Figure 10 shows Specification 6 which is the result of weaving this aspect to Specification 2. Comparing Specification 6 with Specification 4 (Figure 8), the difference resides in whether or not the definition of family *configure* has changed.

Specification 6:

	bf *log*
. . .	
7.	optional_configure = **configure_new** · *log* + 1
. . .	
10.	full_functional_elevator = base_functionality · **configure_new** · *log*
. . .	

Figure 10: Weaving an aspect with a *component* pointcut

Equivalent_component pointcuts refer to the equivalent (or indirect) appearance of the specified product families as components. Taking Specification 1 (Figure 5) as our base specification again, suppose that we want to capture any similar defective behavior that may happen in the family *base_functionality*. Assume that we are not allowed to make changes to the definition of *base_functionality*. In this scenario, we specify an aspect as below:

$$\begin{array}{rl} \text{Aspect} & \text{jp} = \text{jp} \cdot \text{failure\_capture} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{equivalent\_component}(\text{base\_functionality})\Big) \end{array}$$

This pointcut would capture join points whenever *base_functionality* is directly or indirectly referenced. Based on the definition of *base_functionality* given in Line 6 of Specification 1, the join points of this aspect are at Line 7 and Line 8 of Specification 1. Figure 11 shows Specification 7 which is the result of weaving this aspect to Specification 1. Besides the slight difference in meaning, the main difference between Specification 7 and Specification 5 (Figure 9) resides in whether or not the definitions of family *base_functionality* (or its components) have changed.

Specification 7:

. . .

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

. . .

Figure 11: Weaving an aspect with an *equivalent_component* pointcut

3.3.3 Constraint-Related Pointcut

With regard to constraints in PFA specifications, we need to introduce a constraint-related pointcut. As each constraint item consists of three arguments, an extra option in the pointcut is necessary to specify the position where the aspect should be introduced. Therefore, the constraint-related pointcut 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 that is represented by a triple of PFA terms. The keywords are used to specify the *position_list*.

Take Specification 2 (Figure 6) as our base specification. Since the original feature *configure* has been further defined, we intend to extend a "requires" relation to further specify all constraints upon *configure_new*. For example, we can have an aspect as below:

Aspect jp= light_reset

where jp $\in \Big($base, true, *constraint*[*left*](configure_new)$\Big)$

The pointcut would capture the first component of Line 12 in Specification 2 (Figure 6). Figure 12 shows Specification 8 which is the result of weaving this aspect to Specification 2.

Specification 8:

...

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

Figure 12: Weaving an aspect with an *constraint*[*list*] pointcut

Besides default values, the scope of a pointcut and expression of a pointcut also can be specified with a combination of the kind of a pointcut to express specific requirements. We use Specification 1 (Figure 5) as the base specification. Suppose we need to add a feature to capture all defective behaviors with *move_control* in *base_functionality.* A new feature *failure_capture* is required to be included within the product family *elevator_product_line*. This requirement can be expressed as below:

Aspect jp= jp $\cdot$ failure_capture

where jp $\in \Big($*within*(elevator_product_line) ; *cflow*(base_functionality), true, *inclusion*(move_control)$\Big)$

We specify the scope of eligible join points with two scopes *within*(elevator_product_line) and *cflow*(base_functionality). We specify the kind of the pointcut by *inclusion*(move_control). The *within* narrows the scope of join points to only Line 9 of Specification 1 (Figure 5). Moreover, since *cflow* 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. Figure 13 shows Specification 9 which is obtained by weaving this aspect to Specification 1.

3.4 Advice and Aspects in AO-PFA

As discussed in the previous section, *declaration*, *creation* and *component_creation* capture definition join points, while *inclusion*, *component*, *equivalent_component*, and *constraint*[*position_list*] capture reference join points. Therefore, given the kind of pointcuts, an aspect either relates to definition join points or to reference join points. There is a slight difference for specifying aspects that relate to these two types of join points. If the aspects relate to definition join points, *aspectId* should specify new labels that define new product family terms. If the aspects relate to reference join points, *aspectId* should always be expressed as variable *jp* that refers to join points.

Specification 9:

$$\begin{array}{ll}
 & \text{bf } \textit{failure\_capture} \\
\ldots & \\
9. & \text{elevator\_product\_line} = \text{move\_control} + \textbf{base\_functionality} \cdot \textit{failure\_capture} \\
 & \qquad\qquad + \textbf{full\_functional\_elevator} \cdot \textit{failure\_capture} \\
\ldots &
\end{array}$$

Figure 13: Weaving an aspect with scope pointcuts

Additionally, at the requirement level, it is unnecessary to specify the relative introduction time of an aspect against their join points as specified in AspectJ. We discuss and categorize aspects according to the effects of their advice on the 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 to their augmenting, narrowing, and replacing effects upon join points.

Augmenting Aspects: With augmenting aspects, the specification related to join points should still appear in the resulting specification after weaving the aspect. In the proposed language, the behaviour of augmenting aspects can be specified by a product family term constructed with a variable that we denote by *jp*, which is used to represent an instance of the join points. We further classify augmenting aspects as *refine* aspects and *extend* aspects. *Refine* aspects augment the original product families where they are defined, whereas *extend* aspects augment original product families where they are referenced.

Narrowing Aspects: Narrowing aspects result in the absence of original join points in the resulting specification after weaving. Therefore, the behaviour of narrowing aspects corresponds to the constant element 1 of product family algebra, which represents a pseudo-product with no features. Furthermore, *discard* aspects narrow product families or basic features where they are defined, while *disable* aspects narrow product families or basic features where they are referenced.

Replacing Aspects: In this case, after weaving, the original join point is replaced by arbitrary product family terms that do not refer to the original join point. In other words, the behaviour of replacing aspects should be 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 the definition join points and reference join points.

In summary, we categorize aspects as *refine*, *extend*, *discard*, *disable*, *replace*, and *substitute* in the context of product family. Given the syntax of an aspect in AO-PFA, we can directly categorize it according to the form of its advice (i.e., *Aspect(jp)*) and the kind of its pointcut, which will help users to understand the aspect and ease the validation of the aspect.

3.5 Literature Survey Regarding Aspect Oriented Language at the Feature-Modeling Level

AspectJ is the most well known aspect-oriented technique applied at the programming level. Our work mainly adapts several ideas from AspectJ to the product family specification level. Terms in our research are analogous to terms in AspectJ. In contrasting with AspectJ, the feature-related pointcuts *declaration* and *inclusion* can be respectively analogized as the field set and field get pointcuts in AspectJ. The two types of family-related pointcuts *creation* and *component_creation* can be connected to the class/object creation pointcuts in AspectJ. The other two types of family-related pointcuts *component* and *equivalent_component* can be connected to the method-related pointcuts 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.

With regard to the literature of aspect-oriented software development at the early analysis and design stages, most of the existing approaches describe the system with more or less structured natural language specifications, use-case models, scenarios and interaction diagrams. AORE with Arcade [32] is an aspect-oriented requirement engineering approach that modularises and composes the crosscutting concerns at the requirements level to produce an XML requirement specification document. AOSD/UC [19] and the aspectual use-case approach [6] modularise crosscutting functional requirements by extended and included use-cases. Scenario modeling with aspects [41] is a scenario-based technique that models aspectual scenarios by Interaction Pattern Specifications. The main challenge of early aspects is that, unlike at the programming level, specifications of systems tend to be represented as pieces of text-based or graph-based notations making them more difficult to be interpreted and quantified for composing aspects. Consequently, automations of processing specifications and identifying trade-offs early is a non-trivial task. One of the positive efforts for this challenge is to formalise the specifications of systems. Our approach formalises the specification of product families in a programming-like style based on mathematical models, which gives us a more precise and compact representation of a product family.

At the modeling and specification level for product families, many efforts have been taken in the literature to manage the common and variable features. Some of those works are close to our work, which attempts to separate concerns at the early requirement stage and obtain complete specifications by composing different models. Variability Modelling Language for Requirement Engineering (VML4RE) [1] presents a requirement specification language to compose elements from different requirement models. The modeling granularity is at the feature level, which is similar to our technique. The language is based on concrete models that it supports (i.e., use cases, interaction diagrams, and goal models). Xweave [14] is a model weaver supporting the composition of different views. It helps weaving variable parts of architectural models to base models. The weaving mechanism is similar to our technique by specifying pointcuts and advice of aspects. However, it is unable to remove or override existing base model elements. In comparison with VML4RE and

Xweave, our technique is more generic since it is built upon the abstract feature-modeling language of product family algebra [15, 16, 17].

As we have shown, another focus of our approach is formalisation. In [5], a feature algebraic foundation for feature composition is proposed. 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, the approach for feature composition is more related to programming languages. Our approach specificity 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 requirement stage to the implementation stage.

4 Verification of Aspectual composition in AO-PFA

One of the main challenges that need to be tackled in the aspect-oriented paradigm is the verification of aspectual composition. A key characteristic of the aspect-oriented paradigm is the obliviousness of base systems [12]. While this characteristic facilitates the anticipated extension of systems, it makes reasoning on aspect-oriented specifications more difficult. Since aspects are implicitly invoked with the aspectual composition mechanism, verifying the aspectual composition becomes more complicated. Therefore, formal techniques are needed to ensure the correctness of aspectual composition. Most existing tools for their verification are tailored for languages at the detailed design and implementation levels [24]. Until now, related researches are at their early stage and are far from being satisfactory. Moreover, a major type of interference caused by aspectual composition is derived from early development stages and should be handled earlier [35]. However, the main difficulty of verifying aspectual composition at the early analysis and design stage is that the specifications of the considered systems tend to be represented either graphically or in natural language. Unlike at the programming level, the informal or semi-formal specifications make verification on aspectual composition more difficult at the early stages.

In this section, we address the verification of aspectual composition in AO-PFA specifications. Weaving an aspect to a base specification may introduce new specification elements or may modify the base specifications. Intuitively, an aspect should be rejected if it transforms a valid base specification into an invalid one. Instead of verifying the new specifications that result from the composition of aspects, our technique intends to detect invalid aspects w.r.t. the base specification prior to the weaving process.

4.1 Mathematical Setting

4.1.1 Validity Criteria for PFA Specifications

We first establish our mathematical settings to identify what criteria need to be satisfied for a valid PFA specification representing a product family feature model. In PFA specifications, the most basic constructs are those labels that either represent features or product families. Therefore, we abstract validity criteria of PFA specifications at the atomic granularity with regard to those labels. Particularly, we consider properties related to the definitions, references, and dependencies of those basic labels.

Construction 1. *Given a PFA specification S, let M_S be the multi-set of labels that are present in S at basic feature declarations or at the left-hand sides of labeled product families. We call M_S the defining label multi-set associated with the specification S.* □

Definition 5 (Definition-valid specification)**.** *We say that a PFA specification S is definition-valid iff*

$$\forall(v \mid v \in M_S : +(v \mid v \in M_S : 1) = 1)$$

where M_S is the defining label multi-set of S. □

Definition 5 indicates that a specification is definition-valid iff all the elements in $\mathrm{M_S}$ are unambiguously defined labels. Obviously, the multi-set of a definition-valid specification actually forms a set. In this case, we denote the set by $\mathrm{D_S}$, which contains all elements of $\mathrm{M_S}$. Correspondingly, we call $\mathrm{D_S}$ the defining label set associated with a specification S.

Construction 2. *Given a PFA specification S, let R_S be the set of labels that are present in S at the constraints or at the right-hand sides of labeled product families. We call R_S the referencing label set associated with the specification S.* □

Definition 6 (Reference-valid specification)**.** *We say a specification S is reference-valid iff $R_S \subseteq D_S$, where R_S is the referencing label set of S and D_S is the defining label set of S.* □

Definition 6 indicates that a specification is reference-valid iff all the elements in $\mathrm{R_S}$ are defined referencing labels.

Construction 3. *Given a PFA specification S, let D_S be its corresponding defining label set and $G_S = (D_S, E)$ be a digraph. A tuple (u, v) is in E iff u occurs in a product family term T such that the equation $v = T$ is a labeled product family specified in S. We call G_S the label dependency digraph associated with the specification S.* □

Definition 7. *Let $G_S = (V, E)$ be a label dependency digraph associated with a PFA specification S. For $u, v \in V$, we say that u defines v iff*

$$\exists(n \mid n \geq 1 : (u,v)\text{–}path \in E^n)$$

Consequently, we say u and v are mutually defined labels denoted by $mutdef(u, v)$ iff

$$\exists(m, n | m, n \geq 1 : (u,v)\text{–}path \in E^m \wedge (v,u)\text{–}path \in E^n).$$

In particular, if u and v are identical and $m = n = 1$, we say u is self-defined. □

Definition 8 (Dependency-valid specification)**.** *We say that a PFA specification S is dependency-valid iff*

$$\forall(u, v \mid u, v \in D_S : \neg mutdef(u, v))$$

where D_S is the defining label set of S. □

Definition 8 indicates that a valid PFA specification does not have any *mutually defined* or *self-defined* labels.

Lemma 1. *Let $G_S = (V, E)$ be a label dependency digraph of a PFA specification S. Two labels u and v are mutually defined iff there is a cycle including u and v in G_S. In particular, a label u is self-defined iff there is a loop through u in G_S.*

Proof.

u and v are *mutually defined*

$\iff$ $\langle$ Definition 7 $\rangle$

$\exists(m, n \mid m, n \geq 1 : (u, v)\text{–path} \in E^m \wedge (v, u)\text{–path} \in E^n)$

$\iff$ $\langle$ Path concatenation $\rangle$

$\exists(m, n \mid m, n \geq 1 : (u, u)\text{–path} \in E^{m+n} \wedge (v, v)\text{–path} \in E^{m+n})$

$\iff$ $\langle$ Dummy renaming $\rangle$

$\exists(k \mid k \geq 2 : (u, u)\text{–path} \in E^k \wedge (v, v)\text{–path} \in E^k)$

$\iff$ $\langle$ Definition of cycle in digraph $\rangle$

There is a cycle including u and v in $\mathrm{G_S}$

u is self-defined

$\iff$ $\langle$ Definition 7 & $u = v$ & $m = n = 1$ $\rangle$

$\exists(m, n | m = n = 1 : (u, u)\text{–path} \in E^m \wedge (u, u)\text{–path} \in E^n)$

$\iff$ $\langle$ One-point rule & Idempotency of $\wedge$ $\rangle$

(u, u)–path $\in E$

$\iff$ $\langle$ Definition of loop in digraph $\rangle$

There is a loop through u in $\mathrm{G_S}$

□

According to Lemma 1, the digraph $\mathrm{G_S}$ associated with a dependency-valid PFA specification S should be cycle-free and loop-free.

Moreover, since a valid label dependency digraph is a typical digraph that can have a topological ordering, a walk between two vertices is indeed a path. We define a function *Walk: $V \times V \rightarrow$ ordered-list(V)* over a digraph. *Walk*(u, v) returns the list of all vertices along a walk from u to v. In the label dependency digraph, the vertex list *Walk*(u, v) is sufficient to identify a path from u to v. Particularly, if *Walk*(u, v) is empty, it indicates that there is no path from u to v.

4.1.2 Validity Criteria of Aspects in AO-PFA

In AO-PFA, aspects are composed with base specifications at the granularity of product family terms. Therefore, weaving an aspect to a base specification may change the defining label multi-set, the referencing label set, and the label dependency digraph of the original specification. Consequently, an aspect is invalid if it transforms a valid specification to be either definition-invalid, reference-invalid, or dependency-invalid.

Precisely, the effects of weaving an aspect can be abstracted with the following construction.

Construction 4. *Let S' be the new PFA specification obtained by weaving an aspect A to a valid PFA specification S. With respect to S and S', the defining label sets, referencing label sets and dependency digraphs can be constructed according to Constructions 1–3, respectively. To discuss the difference between S' and S, we denote D_A, R_A, E_add_A and E_del_A associated with the aspect A as follows:*

- *Let D_A be a set of labels introduced by A which will be present at basic feature declarations or left-hand sides of labeled product families in S'. As every element $v \in D_A$ is a defining label, the defining label multi-set of S' is $M_{S'} = D_S \sqcup D_A$ where $\sqcup$ denotes the multi-set union. Correspondingly, we denote it by $D_{S'}$ if all elements in $M_{S'}$ occur only once.*
- *Let R_A be a set of labels introduced by A which will be present at constraints or right-hand sides of labeled product families in S'. As every element $v \in R_A$ is a referencing label, the referencing label set of S' is $R_{S'} = R_S \cup R_A$.*
- *Let E_add_A be a set of tuples (u, v) such that u is a label that will be introduced by A at the right-hand side of a labeled product family in S' and v is the label present at the left-hand side of the labeled product family. Let E_del_A be a set of tuples (u, v) such that the label u will be removed by A from the right-hand side of a labeled product family in S, and v is the label present at the left-hand side of the labeled product family. As E_add_A and E_del_A correspond to edge additions and deletions in G_S, the dependency digraph of S' is $G_{S'} = (D_{S'},\ (E_S \cup E\_add_A) - E\_del_A)$.*

□

Detection of Definition-Invalid and Reference-Invalid Aspects in AO-PFA

Definition 9 (Definition-valid aspect)**.** *We say that an aspect A is definition-valid with regard to a specification S iff*

$$D_S \cap D_A = \emptyset$$

□

Definition 9 indicates that a definition-valid aspect would not lead to a potentially ambiguous label definition in the specification after weaving.

Table 1: The effect of pointcut kinds on $\mathrm{D_A}$ and $\mathrm{R_A}$

kind of pointcut	$\mathbf{D_A}$ **contains**	$\mathbf{R_A}$ **contains**
inclusion	all newly introduced labels specified by *Advice(jp)*	all labels specified by *Advice(jp)*
component		
equivalent_component		
declaration	all newly introduced labels specified by *Advice(jp)* and all labels in *aspectId*	all labels specified by *Advice(jp)* and *aspectId*
creation		
component_creation		
constraint[*list*]	empty set	all labels specified by *Advice(jp)*

Definition 10 (Reference-valid aspect)**.** *We say that an aspect A is reference-valid with regard to a PFA specification S iff*

$$(R_S \cup R_A) \;\subseteq\; (D_S \cup D_A) \qquad \square$$

Definition 10 indicates that a valid aspect would not lead to any undefined referencing labels in the specification after weaving.

We construct the defining label set $\mathrm{D_A}$ and the referencing label set $\mathrm{R_A}$ associated with an aspect A to detect definition-invalid and reference-invalid aspects according to Definitions 9 and 10.

Let an aspect A be given in the general form as described in Section 3. The kind of a pointcut decides the position of join points, which affects the construction of $\mathrm{D_A}$ or $\mathrm{R_A}$ for an aspect. As mentioned in the previous section, *declaration*, *creation* and *component_creation* intend to introduce new specifications where the specified product families are defined, while pointcuts of type *inclusion*, *component*, *equivalent_component* and *constraint*[*position_list*] intend to introduce new specifications where the specified product families are referenced. With regard to different kinds of pointcuts, the corresponding $\mathrm{D_A}$ and $\mathrm{R_A}$ are directly constructed as given in Table 1.

In verifying definition-validity and reference-validity of aspects, the main cost is to construct the defining label set and referencing label set of base specifications. Particularly, the complexity is $O(V)$, where V is the number of features in the base specification.

Detection of Dependency-Invalid Aspects in AO-PFA

Definition 11 (Dependency-valid aspect)**.** *Let S' be a PFA specification obtained by weaving an aspect A with a valid specification S. We say that A is dependency-valid w.r.t. S iff S' is dependency-valid.* □

Definition 11 indicates that a valid aspect would not lead to *mutually defined* or *self-defined* labels in the specification obtained by the weaving process. Unlike detecting violations of definition-validity and reference-validity, detecting the violation of dependency-validity of an aspect is not straightforward. Instead of examining all edges in $\mathrm{E\_add_A}$ and $\mathrm{E\_del_A}$, we only consider the edges that may introduce loops or cycles in $\mathrm{G_{S'}}$.

In accordance to the pointcut of an aspect, we use the following construction to identify vertices on the dependency digraph in a base specification, which are directly affected by weaving an aspect.

Construction 5. *Let G_S be a dependency digraph associated with a valid specification S. We denote vertices in G_S with regard to an aspect A as follows:*

- *We denote a vertex in G_S corresponding to the term specified by the kind of the pointcut of A by k. When there is no such vertex, a new vertex is created and named k.*
- *We denote a vertex in G_S corresponding to the label specified by the scope of the pointcut of A by s.*

□

Prop. 1 (Pointcut Kind Condition)**.** *An aspect A is dependency-valid w.r.t. a valid PFA specification S if the kind of pointcut of A is "constraint[list]".*

Proof. The kind of pointcut of A is *constraint*[*list*] $\implies$ ($\mathrm{D_A} = \mathrm{E\_add_A} = \mathrm{E\_del_A} = \emptyset$). We then prove $\mathrm{D_A} = \mathrm{E\_add_A} = \mathrm{E\_del_A} = \emptyset \implies A$ is dependency-valid w.r.t. S.

$$
\begin{aligned}
& A \text{ is dependency-valid w.r.t. } S \\
\Longleftrightarrow \quad & \langle \text{ Definition 11 } \rangle \\
& \forall(u,v \mid u,v \in \mathrm{D_{S'}} : \neg\mathrm{mutdef}(u,v)) \\
\Longleftrightarrow \quad & \langle \text{ Definition 7 } \rangle \\
& \forall(u,v \mid u,v \in (\mathrm{D_S} \cup \mathrm{D_A}) : \neg \exists(m,n \mid m,n \geq 1 : \\
& (u,v)\text{–path} \in (\mathrm{E\_add_A} \cup E_S - \mathrm{E\_del_A})^m \wedge (u,v)\text{–path} \in (\mathrm{E\_add_A} \cup E_S - \mathrm{E\_del_A})^n)) \\
\Longleftrightarrow \quad & \langle \text{ Assumption: } \mathrm{D_A} = \mathrm{E\_add_A} = \mathrm{E\_del_A} = \emptyset \ \rangle \\
& \forall(u,v \mid u,v \in \mathrm{D_S} : \neg \exists(m,n \mid m,n \geq 1 : (u,v)\text{–path} \in (E_S)^m \\
& \qquad \wedge (v,u)\text{–path} \in (E_S)^n)) \\
\Longleftrightarrow \quad & \langle \text{ Definition 7 } \rangle \\
& \forall(u,v \mid u,v \in \mathrm{D_S} : \neg\mathrm{mutdef}(u,v)) \\
\Longleftrightarrow \quad & \langle\ S \text{ is dependency-valid (Definition 8) } \rangle \\
& \mathsf{true}
\end{aligned}
$$

Therefore, due to the transitivity of $\Longrightarrow$, when the type of kinded pointcut is *constraint*[*list*], A is dependency-valid w.r.t. S. □

Prop. 2 (Non-cycle Condition)**.** *Let S be a valid PFA specification and A be an aspect that does not satisfy the pointcut kind condition (Prop. 1). Construct the dependency digraph G_S according to Construction 3 and denote or create the vertex k in G_S according to Construction 5. Then A is dependency-valid w.r.t. S if $\forall(x \mid x \in D_S \cap R_A : Walk(k, x) = \emptyset)$.*

Proof. Let $\mathrm{A_{def}}$ be a set of new labels assigned to a labeled product family where join points are present at their left-hand sides. Let $\mathrm{JP_{def}}$ be the set of all labels of a labeled product family where join points are present at their right-hand sides. Obviously, every join point should be a vertex x such that there is a path from k to x. As mentioned in Section 4.1.2, we consider pointcut kinds in two categories that are respectively related to definitions and references of product families. We say that the kind of a pointcut is definition related if it is *declaration*, *creation* or *component_creation*, whereas the kind of a pointcut is reference related if it is *inclusion*, *component*, *equivalent_component*, or *constraint*[*list*]. Provided the pointcut kind condition is not satisfied, we have the following:

- The kind of pointcut is definition related, which implies the following condition:

$$\begin{aligned} &\mathrm{E\_add_A} = \{(u,v) \mid u \in \mathrm{R_A} \wedge v \in \mathrm{A_{def}}\} \cup \{(u,v) \mid u \in \mathrm{A_{def}} \wedge v \in \mathrm{JP_{def}}\} \\ \wedge\ &\mathrm{E\_del_A} = \{(u,v) \in E_S \mid k \text{ defines } u \wedge v \in \mathrm{JP_{def}}\} \end{aligned}$$

- The kind of pointcut is reference related, which implies the following condition:

$$\begin{aligned} &\mathrm{E\_add_A} = \{(u,v) \mid u \in \mathrm{R_A} \wedge v \in \mathrm{JP_{def}}\} \\ \wedge\ &\mathrm{E\_del_A} = \{(u,v) \in E_S \mid k \text{ defines } u \wedge v \in \mathrm{JP_{def}}\} \end{aligned}$$

The edges of $\mathrm{E\_add_A}$ generates a path from each vertex $u \in \mathrm{R_A}$ to each vertex $v \in \mathrm{JP_{def}}$ in the amended dependency digraph. Moreover, due to the ordering properties of the dependency digraph, edges in $\mathrm{E\_del_A}$ would not be within a path that starts from a vertex in $\mathrm{JP_{def}}$ in the original dependency digraph. According to Lemma 1, A is dependency-invalid w.r.t. S iff

$$\begin{aligned} &\exists(u \mid u \in \mathrm{D_S} : u \in \mathrm{JP_{def}} \wedge u \in \mathrm{R_A}) \vee \\ &\exists(u,v \mid u,v \in \mathrm{D_S} : u \in \mathrm{JP_{def}} \wedge v \in \mathrm{R_A} \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)) \end{aligned} \qquad (1)$$

In (1), the first exstential quantification before $\vee$ indicates there are loops in the amended dependency digraph, while the second existential quantification after $\vee$ indicates there are cycles in the amended dependency digraph.

A is dependency-invalid w.r.t. S

$\iff$ $\langle$ Prop. 1 $\rangle$

(1)

$\iff$ $\langle$ Dummy Nesting & Trading rule $\rangle$

$\exists(u \mid u \in \mathrm{D_S} \wedge u \in \mathrm{R_A} : u \in \mathrm{JP_{def}}) \vee \exists\Big(v \mid v \in \mathrm{D_S} \wedge v \in \mathrm{R_S} : \exists(u \mid u \in \mathrm{JP_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))\Big)$

$\iff$ $\langle$ Dummy renaming & Distributivity of $\exists$ & Axiom of set intersection $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in \mathrm{JP_{def}} \vee \exists(u \mid u \in \mathrm{JP_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))\Big)$

$\iff$ $\langle$ $\mathrm{JP_{def}}$ Property: $x \in \mathrm{JP_{def}} \iff \exists(n \mid n \geq 1 : (k,x)\text{–path} \in (E_S)^n)$ & Trading rule $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \vee \exists(u \mid: \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n) \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))\Big)$

$\iff$ $\langle$ Path concatenation & Dummy renaming & Distributivity of $\wedge$ over $\exists$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \vee \Big(\exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m) \wedge \exists(u \mid u \in \mathrm{D_S} : \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n))\Big)\Big)$

$\Longrightarrow$ $\langle$ Weakening $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \vee \exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m)\Big)$

$\iff$ $\langle$ Dummy renaming & Range split & Idempotency of $\vee$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n)\Big)$

$\iff$ $\langle$ Definition of $\mathrm{Walk}(u,v)$ $\rangle$

$\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \mathrm{Walk}(k,v) \neq \emptyset)$

Therefore,

A is dependency-invalid w.r.t. $S \Longrightarrow \exists(x \mid x \in \mathrm{D_S} \cap \mathrm{R_A} : \mathrm{Walk}(k,x) \neq \emptyset)$

If we take its contrapositive form, we have

$\forall(x \mid x \in \mathrm{D_S} \cap \mathrm{R_A} : \mathrm{Walk}(k,x) = \emptyset) \Longrightarrow A$ is dependency-invalid w.r.t. S

$\square$

From the above proof, we have the following equation:

$$\begin{aligned} &A \text{ is dependency-invalid w.r.t. } S \\ \iff &\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in \mathrm{JP_{def}} \vee \exists(u \mid u \in \mathrm{JP_{def}} : \exists(m \mid m \geq 1 : \\ &(u,v)\text{–path} \in (E_S)^m))\Big) \end{aligned} \tag{2}$$

Definition 12. *We say that an aspect is potentially dependency-invalid if it does not satisfy both the point kind condition (Prop. 1) and the non-cycle condition (Prop. 2).* □

Definition 12 indicates that an aspect is possibly dependency-invalid iff its kind of pointcut is not *constraint*[*list*] and $\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \mathrm{Walk}(k,v) \neq \emptyset))$. Using the above properties, we verify whether an aspect is potentially invalid in accordance to the kind of pointcut. If a potentially invalid aspect is detected, we continue to verify whether it is actually invalid with regard to its base specification in accordance to the scope of its pointcut.

Prop. 3. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. When the scope of the pointcut is "base", A is always dependency-invalid w.r.t. S.*

Proof. When the scope of a pointcut is *base*, join points are where k is present. Therefore, $\mathrm{JP_{def}} = N^+(k)$.

A is dependency-invalid w.r.t. S

$\Longleftrightarrow$ $\langle$ Equation (2) & $\mathrm{JP_{def}} = N^+(k)$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in N^+(k) \vee \exists(u \mid u \in N^+(k) : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)))\big)$

$\Longleftrightarrow$ $\langle$ $x \in N^+(k) \Longleftrightarrow (k,x)\text{–path} \in E_S$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (k,v)\text{–path} \in E_S \vee \exists(u \mid u \in \mathrm{D_S} : (k,u)\text{–path} \in E_S \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))\big)$

$\Longleftrightarrow$ $\langle$ Path concatenation & Distributivity of $\wedge$ over $\exists$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (k,v)\text{–path} \in E_S \vee \big(\exists(u \mid u \in \mathrm{D_S} : (k,u)\text{–path} \in E_S) \wedge \exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m)\big)\big)$

$\Longleftrightarrow$ $\langle$ Prop. 2 does not hold $\Longrightarrow$ $\exists(u \mid u \in \mathrm{D_S} : (k,u)\text{–path} \in E_S)$ & Identity of $\wedge$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (k,v)\text{–path} \in E_S \vee \exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m)\big)$

$\Longleftrightarrow$ $\langle$ One-point rule & Range split $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(m \mid m \geq 1 : (k,v)\text{–path} \in (E_S)^m)\big)$

$\Longleftrightarrow$ $\langle$ Definition of $\mathrm{Walk}(u,v)$ & Precondition: Prop. 2 does not hold $\rangle$

true

□

Prop. 4. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. When the scope of the pointcut is "protect(base)", A is always dependency-valid w.r.t. S.*

Proof. When the scope of a pointcut is *protect(base)*, the set $\mathrm{JP}_{\mathrm{def}}$ is empty.

$$\begin{aligned}
& \text{A is dependency-invalid} \\
\Longleftrightarrow \quad & \langle \text{ Equation (2)} \quad \& \quad \mathrm{JP}_{\mathrm{def}} = \emptyset \ \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ \mathsf{false} \vee \exists(u \mid \mathsf{false} \ : \ \exists(m \mid m \geq 1 \ : \ (u,v)\text{–path} \in (E_S)^m\,)\,)\big) \\
\Longleftrightarrow \quad & \langle \text{ Empty range} \quad \& \quad \exists\text{-False Body: } \exists(x \mid R : \mathsf{false}\,) \iff \mathsf{false} \ \rangle \\
& \mathsf{false}
\end{aligned}$$

□

Prop. 5. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 3 and denote or create the vertices k and s in G_S according Construction 5. When the scope of the pointcut is "within", A is dependency-invalid w.r.t. S iff*

$$\exists(v \mid v \in D_S \cap R_A \ : \ s \in \mathit{Walk}(k,v) \wedge s \neq k\,).$$

Proof. When the scope of a pointcut is *within*, join points are bound to a labeled product family whose label is s. Therefore, we have $\mathrm{JP}_{\mathrm{def}} = s$. Besides, there should be a path form k to s.

$$\begin{aligned}
& A \text{ is dependency-invalid w.r.t. } S \\
\Longleftrightarrow \quad & \langle \text{ Equation (2)} \quad \& \quad \mathrm{JP}_{\mathrm{def}} = s \quad \& \quad \text{There is a path from } k \text{ to } s \ \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ v = s \vee \exists(u \mid u = s \ : \ \exists(m \mid m \geq 1 \ : \ (u,v)\text{–path} \in \\
& (E_S)^m\,)\,)\big) \wedge \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,) \\
\Longleftrightarrow \quad & \langle \text{ Distributivity of } \wedge \text{ over } \exists \quad \& \quad \text{One-point rule } \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ (v = s \vee \exists(m \mid m \geq 1 \ : \ (s,v)\text{–path} \in (E_S)^m\,)) \wedge \exists(n \mid \\
& n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)\big) \\
\Longleftrightarrow \quad & \langle \text{ Definition of Walk}(u,v) \ \& \ S \text{ is dependency-valid} \iff \text{Walk}(x,x) = \emptyset \ \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ (s = v \vee (\text{Walk}(s,v) \neq \emptyset \ \wedge \ s \neq v)) \wedge (\text{Walk}(k,s) \neq \emptyset \wedge k \neq s)\big) \\
\Longleftrightarrow \quad & \langle \text{ Absorbing: } p \vee (q \wedge \neg p) \iff (p \vee q) \ \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ (s = v \vee \text{Walk}(s,v) \neq \emptyset) \wedge \text{Walk}(k,v) \neq \emptyset \wedge k \neq s\big) \\
\Longleftrightarrow \quad & \langle \text{ Path concatenation } \rangle \\
& \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ s \in \text{Walk}(k,v) \wedge k \neq s\big)
\end{aligned}$$

□

Prop. 6. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 3 and denote or create the vertex k in G_S according Construction 5. When the scope of the pointcut is "protect(within)", A is dependency-invalid w.r.t. S iff*

$$\exists(v \mid v \in D_S \cap R_A \;:\; s \notin \mathit{Walk}(k,v) \vee s = k\,).$$

Proof. When the scope of a pointcut is *protect(within)*, if there is a path from k to s, labels in $\mathrm{JP_{def}}$ should exclude s. Otherwise, the set $\mathrm{JP_{def}}$ is identical with the one specified by pointcut of type *base*.

A is dependency-invalid w.r.t. S

$\iff$ $\langle$ Equation (2) & $\mathrm{JP_{def}}$ properties & Prop. 3 $\rangle$

$\Big(\exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k\,) \wedge \exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\,\exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n\,) \wedge v \neq s) \vee \exists(u \mid \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n\,) \wedge u \neq s : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m\,)\,)\Big)\Big) \vee (\neg\,\exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k\,) \wedge \mathsf{true})$

$\iff$ $\langle$ Identity of $\wedge$ & Absorbing: $(p \wedge q) \vee \neg p \iff q \vee \neg p$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\,\exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n\,) \wedge v \neq s) \vee \exists(u \mid \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n\,) \wedge u \neq s : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m\,)\,)\Big) \vee \neg\,\exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k\,)$

$\iff$ $\langle$ Trading rule & Definition of $\mathrm{Walk}(u,v)$ & Path concatenation $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\mathrm{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee \exists(u \mid u \neq s : u \in \mathrm{Walk}(k,v) \wedge u \neq k \wedge u \neq v\,)\Big) \vee \mathrm{Walk}(k,s) = \emptyset$

$\iff$ $\langle$ Trading rule & Generalized De Morgan $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\mathrm{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee \neg\,\forall(u \mid u = s : u \in \mathrm{Walk}(k,v) \wedge u \neq k \wedge u \neq v\,)\Big) \vee \mathrm{Walk}(k,s) = \emptyset$

$\iff$ $\langle$ One-point rule & De Morgan $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\mathrm{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee s \notin \mathrm{Walk}(k,v) \vee s = k \vee s = v)\Big) \vee \mathrm{Walk}(k,s) = \emptyset$

$\iff$ $\langle$ Distributivity of $\vee$ over $\wedge$ & Excluded Middle & Absorbing: $p \wedge (p \vee q) \iff p$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v) \vee s = k)\Big) \vee \mathrm{Walk}(k,s) = \emptyset$

$\iff$ $\langle$ $\mathrm{Walk}(k,s) = \emptyset \implies \exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v)\,)$ & Distributivity of $\vee$ over $\exists$ & $(p \implies q) \implies ((p \vee q) \equiv q)$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v) \vee s = k)\Big)$

□

Prop. 7. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 3 and denote or create the vertices k and s in G_S according Construction 5. When the scope of the pointcut is "cflow", A is dependency-invalid w.r.t. S iff*

$$\exists(v \mid v \in D_S \cap R_A \ : \ s \in \mathit{Walk}(k,v) \wedge s \neq k \wedge s \neq v\,).$$

Proof. When the scope of a pointcut is *cflow*, join points are bound to a labeled product family where s is present at their right-hand sides. Therefore, $\mathrm{JP_{def}} = N^+(s)$. Besides, there should be a path from k to s.

A is dependency-invalid w.r.t. S

$\iff$ $\langle$ Equation (2) & $\mathrm{JP_{def}} = N^+(s)$ & There is a path from k to s $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ v \in N^+(s) \vee \Big(\exists(u \mid u \in N^+(s) \ : \ \exists(m \mid m \geq 1 \ : \ (u,v)\text{–path} \in (E_S)^m\,)\,)\Big) \wedge \ \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)$

$\iff$ $\langle$ $v \in N^+(s) \iff (s,v)\text{–path} \in E_S$ & Distributivity of $\wedge$ over $\exists$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ ((s,v)\text{–path} \in E_S \vee \ \exists(u \mid: \ (s,u)\text{–path} \in E_S \wedge \ \exists(m \mid m \geq 1 \ : (u,v)\text{–path} \in (E_S)^m\,)\,)) \wedge \ \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)\Big)$

$\iff$ $\langle$ Path concatenation & Dummy renaming & $(p \implies q) \implies (p \wedge q \equiv p)$ $\rangle$

$\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ ((s,v)\text{–path} \in E_S \vee \ \exists(m \mid m \geq 2 \ : \ (s,v)\text{–path} \in (E_S)^m\,)) \wedge \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)\,)$

$\iff$ $\langle$ One-point rule & Range split $\rangle$

$\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ \exists(m \mid m \geq 1 \ : \ (s,v)\text{–path} \in (E_S)^m\,) \wedge \ \exists(n \mid n \geq 1 \ : (k,s)\text{–path} \in (E_S)^n\,)\,)$

$\iff$ $\langle$ Path concatenation & Definition of $\mathrm{Walk}(u,v)$ & S is dependency-valid $\iff \mathrm{Walk}(x,x) = \emptyset$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \ : \ s \in \mathrm{Walk}(k,v) \wedge s \neq v \wedge k \neq s\Big)$

□

Prop. 8. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 3 and denote or create the vertices k and s in G_S according Construction 5. When the scope of the pointcut is "protect(cflow)", A is dependency-invalid w.r.t. S iff*

$$\exists(v \mid v \in D_S \cap R_A \ : \ s \notin \mathit{Walk}(k,v) \vee s = k \vee s = v\,).$$

Proof. When the scope of a pointcut is protect(*cflow*), if there is a path from k to s, labels in $\mathrm{JP}_{\mathrm{def}}$ should not include successors of s. Otherwise, the set $\mathrm{JP}_{\mathrm{def}}$ is identical with the one specified by a pointcut with scope *base*.

A is dependency-invalid w.r.t. S

$\iff$ $\langle$ Equation (2) & $\mathrm{JP}_{\mathrm{def}}$ properties & Prop. 3 $\rangle$

$\Big(\exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \wedge v \notin N^+(s) \vee \exists(u \mid \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n) \wedge u \notin N^+(s) : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)) \big) \vee (\mathsf{true} \wedge \neg \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k))$

$\iff$ $\langle$ Distributivity of $\wedge$ over $\exists$ & Distributivity of $\wedge$ over $\vee$ & $x \in N^+(s) \iff (s,x)\text{–path} \in E_S$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge (s,v)\text{–path} \notin E_S) \vee \exists(u \mid \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge (s,u)\text{–path} \notin E_S : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)) \big) \vee \neg \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k)$

$\iff$ $\langle$ Trading rule & Generalized De Morgan $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge (s,v)\text{–path} \notin E_S) \vee \neg \forall(u \mid (s,u)\text{–path} \in E_S : \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n) \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)) \big) \vee \neg \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k)$

$\iff$ $\langle$ Trading rule & Path concatenation $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k) \wedge (s,v)\text{–path} \notin E_S) \vee \neg(\exists(n \mid n \geq 1 : (k,s)\text{–path} \in (E_S)^n) \wedge \exists(m \mid m \geq 2 : (s,v)\text{–path} \in (E_S)^m)) \big) \vee \neg \exists(k \mid k \geq 1 : (k,s)\text{–path} \in (E_S)^k)$

$\iff$ $\langle$ Definition of $\mathrm{Walk}(u,v)$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\mathrm{Walk}(k,v) \neq \emptyset \wedge k \neq s \wedge v \notin N^+(s)) \vee \neg(s \in \mathrm{Walk}(k,v) \wedge k \neq s \wedge s \neq v \wedge v \notin N^+(s) \big) \vee \mathrm{Walk}(k,s) = \emptyset$

$\iff$ $\langle$ De Morgan $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : (\mathrm{Walk}(k,v) \neq \emptyset \wedge k \neq s \wedge v \notin N^+(s)) \vee s \notin \mathrm{Walk}(k,v) \vee s = k \vee s = v \vee v \in N^+(s) \big) \vee \mathrm{Walk}(k,s) = \emptyset)$

$\iff$ $\langle$ Distributivity of $\vee$ over $\wedge$ & Excluded Middle & Absorbing $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v) \vee s = k \vee s = v \big) \vee \mathrm{Walk}(k,s) = \emptyset)$

$\iff$ $\langle$ $\mathrm{Walk}(k,s) = \emptyset \implies \exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v)\big)$ $\rangle$

$\exists\big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \notin \mathrm{Walk}(k,v) \vee s = k \vee s = v \big)$

□

Prop. 9 (Dependency-invalid aspect). *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Let a be a vertex that invalidates the condition of Prop. 2. Vertices k and s are denoted or created in G_S according to A as prescribed in Construction 5. Provided the set of join points is nonempty, the aspect A is dependency-invalid w.r.t. S if Dep_invalid(ts), where ts represents the scope of the pointcut and*

$$\mathit{Dep\_invalid}(ts) \overset{\mathsf{def}}{\Leftrightarrow} \begin{cases} \mathsf{true} & \text{if } ts \text{ is } base \\ s \in \mathit{Walk}(k,a) \wedge s \neq k & \text{if } ts \text{ is } within \\ s \in \mathit{Walk}(k,a) \wedge s \neq k \wedge s \neq a & \text{if } ts \text{ is } cflow \\ \neg \mathit{Dep\_invalid}(ts') & \text{if } ts \text{ is } protect(ts') \\ \mathit{Dep\_invalid}(ts_1) \vee \mathit{Dep\_invalid}(ts_2) & \text{if } ts \text{ is } (ts_1 : ts_2) \\ \mathit{Dep\_invalid}(ts_1) \wedge \mathit{Dep\_invalid}(ts_2) & \text{if } ts \text{ is } (ts_1 \,;\, ts_2) \end{cases}$$

Proof.

(1) $ts = base$

A is dependency-invalid w.r.t. S
$\Longleftrightarrow$ ⟨ Prop. 3 & a is a vertex associated to A that invalidates Prop. 2 ⟩
true

(2) $ts = within$

A is dependency-invalid w.r.t. S
$\Longleftrightarrow$ ⟨ Prop. 5 ⟩
$\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \in \mathrm{Walk}(k,v) \wedge s \neq k)$.
$\Longleftarrow$ ⟨ Witness: a is a vertex associated to A that invalidates Prop. 2 ⟩
$s \in \mathrm{Walk}(k,a) \wedge s \neq k$

(3) $ts = cflow$

A is dependency-invalid w.r.t. S
$\Longleftrightarrow$ ⟨ Prop. 7 ⟩
$\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : s \in \mathrm{Walk}(k,v) \wedge s \neq k \wedge s \neq v)$
$\Longleftarrow$ ⟨ Witness: a is a vertex associated to A that invalidates Prop. 2 ⟩
$s \in \mathrm{Walk}(k,a) \wedge s \neq k \wedge s \neq a$

(4) ts=protect(ts')

- ts'=$base$

  A is dependency-invalid w.r.t. S
  $\Longleftrightarrow$ $\langle$ Prop. 4 $\rangle$
  false
  $\Longleftrightarrow$ $\langle$ Proof item (1) $\rangle$
  $\neg$Dep_invalid($base$)

- ts'=$within$

  A is dependency-invalid w.r.t. S
  $\Longleftrightarrow$ $\langle$ Prop. 6 $\rangle$
  $\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \;:\; s \notin \mathrm{Walk}(k, v) \vee s = k\,)$
  $\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Prop. 2 $\rangle$
  $s \notin \mathrm{Walk}(k, a) \vee s = k$
  $\Longleftrightarrow$ $\langle$ Proof item (2) $\rangle$
  $\neg$Dep_invalid($within$)

- ts'=$cflow$

  A is dependency-invalid w.r.t. S
  $\Longleftrightarrow$ $\langle$ Prop. 8 $\rangle$
  $\exists(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \;:\; s \notin \mathrm{Walk}(k, v) \vee s = k \vee s = v\,)$
  $\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Prop. 2 $\rangle$
  $s \notin \mathrm{Walk}(k, a) \vee s = k \vee s = a$
  $\Longleftrightarrow$ $\langle$ Proof item (3) $\rangle$
  $\neg$Dep_invalid($cflow$)

(5) $ts = (ts_1 : ts_2)$
Let the set of join points selected by ts_1 be $\mathrm{JP}^1_{\mathrm{def}}$ and the set of join points selected by ts_2 be $\mathrm{JP}^2_{\mathrm{def}}$, then the set $\mathrm{JP_{def}} = \mathrm{JP}^1_{\mathrm{def}} \cup \mathrm{JP}^2_{\mathrm{def}}$.

A is dependency-invalid w.r.t. S
$\Longleftrightarrow$ $\langle$ Equation (2) & $\mathrm{JP_{def}} = \mathrm{JP}^1_{\mathrm{def}} \cup \mathrm{JP}^2_{\mathrm{def}}$ $\rangle$
$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \;:\; v \in \mathrm{JP}^1_{\mathrm{def}} \cup \mathrm{JP}^2_{\mathrm{def}} \vee\; \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}} \cup \mathrm{JP}^2_{\mathrm{def}} \;:\; \exists(m \mid m \geq 1 \;:$
(u, v)–path $\in (E_S)^m\,)\,)\Big)$
$\Longleftrightarrow$ $\langle$ Range split for idempotent operator $\exists$ $\rangle$
$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} \;:\; v \in \mathrm{JP}^1_{\mathrm{def}} \cup \mathrm{JP}^2_{\mathrm{def}} \vee\; \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}} \;:\; \exists(m \mid m \geq 1 \;:$

(u,v)–path $) \in (E_S)^m\,) \vee \exists(u \mid u \in \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)$–path $\in$ $(E_S)^m\,)\,)\Big)$

$\Longleftrightarrow$ $\langle$ Union set axiom & Associativity and Symmetry of $\vee$ & Distributivity of $\exists$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in \mathrm{JP}^1_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}} : \exists(m \mid m \geq 1 :$ (u,v)–path $\in (E_S)^m\,)\,)\Big) \vee \exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in \mathrm{JP}^2_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{JP}^2_{\mathrm{def}} :$ $\exists(m \mid m \geq 1 : (u,v)$–path $\in (E_S)^m\,)\,)\Big)$

$\Longleftarrow$ $\langle$ Equation (2) & Proof items (1), (2), (3) and (4) $\rangle$

Dep_invalid$(ts_1) \vee$ Dep_invalid(ts_2)

(6) $ts = (ts_1\ ;\ ts_2)$

Let the set of join points selected by ts_1 be $\mathrm{JP}^1_{\mathrm{def}}$ and the set of join points selected by ts_2 be $\mathrm{JP}^2_{\mathrm{def}}$, then the set $\mathrm{JP}_{\mathrm{def}} = \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}}$, provided that $\mathrm{JP}_{\mathrm{def}} \neq \emptyset$.

A is dependency-invalid w.r.t. S

$\Longleftrightarrow$ $\langle$ Equation (2) & $\mathrm{JP}_{\mathrm{def}} = \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}}$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : v \in \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} \vee \exists(u \mid$ $u \in \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)$–path $\in (E_S)^m\,)\,)\Big)$

$\Longleftrightarrow$ $\langle$ Set intersection axiom & Distributivity of $\vee$ over $\wedge$ $\rangle$

$\exists\Big(v \mid v \in \mathrm{D_S} \cap \mathrm{R_A} : \Big(v \in \mathrm{JP}^1_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}}$ $\cap \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)$–path $\in (E_S)^m\,)\,)\Big)$ $\wedge\Big(v \in \mathrm{JP}^2_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)$–path $\in (E_S)^m\,)\,)\Big)\Big)$

$\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Prop. 2 $\rangle$

$\Big(a \in \mathrm{JP}^1_{\mathrm{def}} \vee \exists(u \mid u \in u \in \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,a)$–path $\in (E_S)^m\,)\,)\Big)$ $\wedge\Big(a \in \mathrm{JP}^2_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,a)$–path $\in (E_S)^m\,)\,)\Big)$

$\Longleftarrow$ $\langle$ $\mathrm{JP}^1_{\mathrm{def}} \cap \mathrm{JP}^2_{\mathrm{def}} \neq \emptyset$ & Witness $\rangle$

$\Big(a \in \mathrm{JP}^1_{\mathrm{def}} \vee \exists(m \mid m \geq 1 : (u,a)$–path $\in (E_S)^m\,)\Big) \wedge \Big(a \in \mathrm{JP}^2_{\mathrm{def}} \vee \exists(m \mid m \geq 1 :$ (u,a)–path $\in (E_S)^m\,)\Big)$

$\Longleftrightarrow$ $\langle$ Equation(2) & Proof item (1), (2), (3), and (4) $\rangle$

Dep_invalid$(ts_1) \wedge$ Dep_invalid(ts_2)

$\square$

With regard to the previous constructions, definitions, and propositions, besides constructing the dependency digraph, the most costly process is to find a walk between two vertices in the digraph. Finding a walk between vertices in such digraphs can be achieved by classic graph algorithms with time complexity linear in the size of the digraph. Therefore, verifying the dependency-validity of aspects with the proposed technique has a complexity of $O(V + E)$, where V and E are the number of vertices and edges in the dependency digraph of base specifications. In other words, V is the number of features in the product families, and E is equal to V^2 at the most.

4.1.3 Related Work to Aspects Verification

In the literature of the aspect-oriented paradigm, various aspect-oriented techniques have been proposed to identify and represent aspects at the early requirement and design stages [19, 32, 41]. Most of those approaches are informal and their support of validation and verification are limited. Those informal approaches are easy to understand and are suited for user validation. But the verification in those approaches is only accomplished by informally "walking through" the artefacts [8]. On the other hand, the language AO-PFA provides a formal and compact way to modularise and compose aspects in product family specifications. The mathematical and formal nature of AO-PFA eases the formal verification of aspectual composition as discussed in this paper.

With regard to the formal verification of aspectual composition, model checking (e.g., [23, 26]) and code static analysis (e.g., [7, 36, 37]) are mainly used in existing approaches at the programming level. However, due to complex notations that are needed for the expressiveness of a programming language, most of those researches focus on detecting interferences caused by one or some types of aspectual composition. Although analogised from AspectJ [25], the notations used in AO-PFA are simplified and unified with the help of product family algebra. In this paper, we intend to detect interferences caused by all types of aspectual composition in AO-PFA.

Our research is inspired by a static code analysis approach described in [34] that characterizes the direct and indirect interactions of aspects with base systems. By considering aspects and base systems in the context of AO-PFA, the interaction classification, namely, orthogonal, independent, observation, actuation, and interference, help us to extract all necessary validity criteria (i.e., definition-validity, reference-validity and dependency-validity) for specifications and aspects in our approach.

4.2 Case Study: Home Automation Family

We illustrate our approach with a case study of a home automation product line adapted from [31]. Basically, a home automation system includes control devices, communication networks, user interfaces and a home gateway. Different types of devices, network standards, and user interfaces can be selected for different products. A home gateway offers different services for overall system management.

To shorten expressions, we use the abbreviations as shown in Table 2. Figure 14 specifies the feature model of the home automation product line. Figure 15 is a house configuration based on the home automation product line, i.e., every configuration should be subfamily of *Home_Automation_product_line* and satisfy the constraints specified in Figure 14. We are going to illustrate how a base specification can be amended using aspects. The specification

Table 2: Basic Features and Families Abbreviation

Feature	**Abbreviation**	**Feature**	**Abbreviation**
time_controller	tim_con	*luminance_sensor*	lum_sen
weather_sensor	wea_sen	*graphical_tv*	gra_tv
web_based	web_bas	*PDA*	PDA
cable	cable	*wireless*	wireless
Internet	inter	*mobile_phone_network*	mob_net
manual_lighting	man_lig	*smart_lighting*	sma_lig
manual_door_and_window	man_daw	*smart_door_and_window*	sma_daw
heating_system	heat_sys	*stereo_system*	ster_sys
Light_device	Lig_dev	*Door_and_window_device*	Daw_dev
User_Interface	Usr_Int	*Communication*	Commun
Lighting_control	Lig_con	*Door_and_window_control*	Daw_con
Home_appliance_control	HApp_con	*Home_gateway*	Home_gateway
Home_Automation_product_line	Home_Auto_PL		

in Figure 14 is used in Case 1 and Case 2 as our base specification, while the specification in Figure 15 is used as our base specification in Case 3.

4.2.1 Specifying Aspects in AO-PFA

Case 1: A new device is added to the product family

An electronic door lock is controlled by the *sma_daw* to open the door. Basically, it uses a *tim_con* to decide when to open and close a door. Later, a new finger printer reader device, abbreviated as *fin_reader*, is available. The feature *sma_daw* uses the feature *fin_reader* for the authentication mechanism instead of the feature *tim_con* to control the door. We write several aspects to specify changes to the base specification, which correspond to a sequence of activities for introducing the feature *fin_reader* into the product family.

(a) We intend to deploy new fingerprint reader device in the product family. We use a family-related *creation* pointcut to capture the left-hand side of the labeled product family *Daw_dev* in Specification *home_automation_product_line* (i.e., Line 19 in Figure 14), and specify the aspect as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp\_new= jp} \cdot (1+\text{fin\_reader}) \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{creation}(\text{Daw\_dev})\Big)\end{array}$$

The *creation* pointcut refers to join points at the exact definitions of labeled product families. The body of the advice is a product family term with the variable *jp*, which indicates an augmenting aspect. According to the aspect classification described in Section 3.4, the above aspect is a *refine* aspect. In other words, we compose an aspect

Specification *home_automation_product_line*:

```
1.   bf tim_con
2.   bf lum_sen
3.   bf wea_sen
4.   bf gra_tv
5.   bf web_bas
6.   bf PDA
7.   bf cable
8.   bf wireless
9.   bf inter
10.  bf mob_net
11.  bf man_lig
12.  bf sma_lig
13.  bf man_daw
14.  bf sma_daw
15.  bf heat_sys
16.  bf ster_sys
17.  bf wat_intr
     % Device Control
18.  Lig_dev = 1+lum_sen
19.  Daw_dev = tim_con· (1+wea_sen)
     % User Interface
20.  Usr_Int = gra_tv· (1+web_bas) · (1+PDA)
     % Communication
21.  Commun = (cable+wireless+cable· wireless) · (1+inter) · (1+mob_net)
     % Application Programmes
22.  Lig_con = man_lig· (1+sma_lig)
23.  Daw_con = man_daw· (1+sma_daw)
24.  HApp_con = (1+heat_sys) · (1+ster_sys) · (1+wat_intr)
25.  Home_gateway = Lig_con· Daw_con· HApp_con)
     % Home_Automation_product_line
26.  Home_Auto_PL = Commun· Usr_Int· Lig_dev· Daw_dev· Home_gateway
     % Constraints
27.  constraint(sma_lig, Home_Auto_PL, Lig_dev)
28.  constraint(sma_daw, Home_Auto_PL, Daw_dev)
29.  constraint(web_bas, Home_Auto_PL, inter)
```

Figure 14: The Base Specification for Case 1 and Case 2

Specification *home_automation_configuration*:

```
     import home_automation_product_line.spec
     %Predefined configuration:
1.   Economic_mode = (lum_sen· tim_con) · cable· gra_tv· (man_lig· sma_lig
                        · man_daw· sma_daw· heat_sys)
     % Customized configuration
2.   kitchen = Economic_mode
3.   Living_room = Economic_mode· ster_sys
4.   Bath_room = tim_con· cable· gra_tv· (man_lig· sma_daw· heat_sys· wat_intr)
5.   Bed_room = (lum_sen· tim_con) · (cable· wireless· mob_net) · gra_tv
                  · ster_sys· heat_sys· man_lig· sma_lig· man_daw· sma_daw
6.   Reading_room = (lum_sen· tim_con) · (cable· wireless· inter· mob_net) ·
                  (gra_tv· web_bas) · (man_lig· sma_lig· man_daw· sma_daw· heat_sys)
7.   House = kitchen· Living_room· Bath_room· Bed_room· · Reading_room
```

Figure 15: The Base Specification for Case 3

with the base specification to *refine* the family of door and window devices.

(b) The next step is to replace the original basic feature *sma_daw* with a new one, e.g., *sma_daw_new*, which has interactions with the database and the finger print reader. We use a feature-related *declaration* pointcut to capture the basic feature declaration of *sma_daw* in Specification *home_automation_product_line* (i.e., Line 14 in Figure 14), and specify an aspect as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp\_new}= \text{sma\_daw\_with\_fin\_reader}\\ \text{where} & \text{jp} \in \Big(\text{base, true, } \mathit{declaration}(\text{sma\_daw})\Big)\end{array}$$

The *declaration* pointcut refers to join points at the definitions of basic features and the body of the advice is a ground product family term. Based on the aspect classification mechanism again, we compose an aspect to *replace* the original smart door and window feature.

(c) After a while, the *fin_reader* mechanism is ensured to be more suitable than the *tim_con* for the smart control of door. We use a *declaration* pointcut again to capture the basic feature declaration of *tim_con* in Specification *home_automation_prod–uct_line* (i.e., Line 1 in Figure 14), and specify an aspect as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp\_new}= 1\\ \text{where} & \text{jp} \in \Big(\text{base, true, } \mathit{declaration}(\text{tim\_con})\Big)\end{array}$$

The above poincut is related to definitions of a basic feature, and the body of the advice is element 1, which indicates a narrowing aspect. Thus, we specify a *discard* aspect to completely remove the basic feature *tim_con* from the product line.

Case 2: A new functionality is introduced in the product family

A sequence of activities are scheduled in the *Home_gateway* for fire detection. This new functionality, abbreviated as *fir_det_flow* would interact with *sma_lig*, *sma_daw* and *HApp_con* features [31]. Additional devices, *fire_sen* (fire_sensor) and *alarm* are required as well. The aspects below can be used to specify this scenario.

(a) The new functionality *fir_det_flow* can be added to the product family *Home_gateway* when certain other features are available. We use the scope *within* to narrow the join points within the textual structure of *Home_gateway* (i.e., Line 25 in Figure 14). Moreover, we use a family-related *component* pointcut to capture product families where *sma_lig*, *sma_daw* and *HApp_con* are all referenced. An aspect is specified as follows:

$$\begin{array}{rl} \text{Aspect} & \text{jp= jp} \cdot \text{fir\_det\_flow} \\ \text{where} & \text{jp} \in \Big(\textit{within}(\text{Home\_gateway}),\ \text{true}, \textit{component}(\text{sma\_lig}\cdot\text{sma\_daw}\cdot\ \text{HApp\_con})\Big) \end{array}$$

The *component* pointcut refers to join points where specified product families are referenced. Apparently, the advice of the above aspect has augmenting effects. Based on our classification mechanism, we use the above aspect to *extend* the original home gateway with new functionality.

(b) Next, we want to replace the basic feature *fir_det_flow* by a product family that consists of all necessary interactions with other product families. The original declaration of the basic feature *fir_det_flow* is again captured with a poincut *declaration* and we specify another *replace* aspect as follows:

$$\begin{array}{rl} \text{Aspect} & \text{jp\_new= sma\_lig}\cdot\ \text{sma\_daw}\cdot\ \text{HApp\_con} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{declaration}(\text{fir\_det\_flow})\Big) \end{array}$$

(c) New devices *fire_sen* and *alarm* are deployed to enable the fire detection functionality. Consequently, new operations to cooperate with *fire_sen* and *alarm* should be added. We use a family-related *component_creation* pointcut and specify a *refine* aspect as follow:

$$\begin{array}{rl} \text{Aspect} & \text{jp\_new= jp} \cdot \textit{fire\_sen} \cdot \textit{alarm} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{component\_creation}(\text{fir\_det\_flow\_new})\Big) \end{array}$$

Case 3: Support variabilities of the product family

The product family supports the variability of user interfaces. For instance, the optional feature *PDA* should be able to be included or excluded with different configurations. We specify several aspects below to exemplify the support of variabilities.

(a) We want to upgrade all configurations that have all features in *Economic_mode* with the interface *PDA* in Specification $home\_automation\_configuration$. We use an *equivalent_component* pointcut to capture the join points at the right-hand sides of Line 5, Line 6 and Line 7 in Figure15 and specify an *extend* aspect as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp= jp} \cdot \text{PDA} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{equivalent\_component}(\text{Economic\_mode})\Big)\end{array}$$

(b) Assume that the *PDA* interface requires the support of the *wireless* network. For economic reasons, we use *wireless* instead of *cable* for the communication. We use a feature-related *inclusion* pointcut to capture *cable* at the right-hand sides of Line 4, Line 5 and Line 6 in Figure 15. An aspect is specified as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp= wireless} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \textit{inclusion}(\text{cable})\Big)\end{array}$$

The *inclusion* pointcut refers to join points where basic features are referenced. The body of advice is in the form of a ground term, which identifies a replacement effect. Therefore, we categorize the above aspect as a *substitute* aspect. By composing this aspect, we *substitute* cable communication with wireless communication in all customized home automation configurations.

(c) Suppose that the *PDA* interface cannot operate correctly in the *Reading_room* for some unknown reason. We again use *within* to specify the scope of join points within Line 6 in the specification of Figure 15, and use the *inclusion* pointcut to capture *PDA* at the right-hand side of Line 6. An aspect is specified as follows:

$$\begin{array}{rl}\text{Aspect} & \text{jp= 1} \\ \text{where} & \text{jp} \in \Big(\textit{within}(\text{Reading\_room}), \text{ true, } \textit{inclusion}(\text{PDA})\Big)\end{array}$$

The above pointcut is related to the references of a basic feature and the advice indicates a narrowing effect. Therefore, the above aspect is categorized as a *disable* aspect. We can *disable* the PDA interface in the reading room by compositing this aspect.

4.2.2 Verifying Aspectual Composition in AO-PFA

Verification of definition-validity For example, we take the aspect of Case 1(a) to illustrate the evaluation of definition-validity of an aspect w.r.t. its base specification. Using Construction 1, the defining label multi-set of the base Specification *home_automation* is as follows:

$\mathrm{M}_{\text{home\_automation\_product\_line}}$={ *tim_con*, *lum_sen*, *wea_sen*, *gra_tv*, *web_bas*, *PDA*, *cable*, *wireless*, *inter*, *mob_net*, *man_lig*, *sma_lig*, *man_daw*, *ster_sys*, *sma_dawheat_sys*, *wat_intr*, *Lig_dev*, *Daw_dev*, *Usr_Int*, *Commun*, *Lig_con*, *HApp_con*, *Home_gateway*, *Home_Auto_PL*, *Daw_con*}

With regard to Definition 5, we claim that the base Specification *home_automation* is definition-valid. Consequently,

$$\mathrm{D}_{\text{home\_automation\_product\_line}} = \mathrm{M}_{\text{home\_automation\_product\_line}}.$$

With regard to the definition-validity of the aspect of Case 1(a), we construct the defining label set of the aspect according to the row corresponding to *creation* in Table 1 and obtain

$$\mathrm{D}_{\mathrm{case1a\_aspect}} = \{\mathit{fin\_reader}, \mathit{Daw\_dev\_new}\}.$$

The intersection of $\mathrm{D}_{\mathrm{home\_automation}}$ and $\mathrm{D}_{\mathrm{case1a\_aspect}}$ is empty. Therefore, the considered aspect is definition-valid w.r.t. the Specification *home_automation_product_line* according to Definition 9.

We use another example to show how to detect certain conflicts among aspects using the proposed criteria. Let Specification *home_automation_one* be the resulting specification after weaving the aspect of Case $1(a)$. We can construct the defining label set of Specification *home_automation_one* in accordance to Construction 1. Particularly,

$$\mathrm{D}_{\mathrm{home\_automation\_one}} = \mathrm{D}_{\mathrm{home\_automation\_product\_line}} \sqcup \mathrm{D}_{\mathrm{case1a\_aspect}}.$$

An aspect is specified below to include a new device *fin_reader* as a mandatory feature in the subfamily *Daw_dev*:

$$\begin{array}{rl} \text{Aspect} & \text{jp\_new= jp} \cdot \text{fin\_reader} \\ \text{where} & \text{jp} \in \Big(\text{base, true, } \mathit{creation}(\text{Daw\_dev})\Big) \end{array}$$

According to Table 1, the defining label set of the above aspect is $\{\mathit{Daw\_dev\_new}\}$, whose intersection with $\mathrm{D}_{\mathrm{home\_automation\_one}}$ is nonempty. Thus, this aspect is definition-invalid w.r.t. the Specification *home_automation_one*, which indicates that weaving such an aspect will cause a definition-invalid specification.

Verification of reference-validity We continue to illustrate the evaluation of reference-validity of an aspect w.r.t. its base specification with the example of Case $1(a)$. The referencing label set of the Specification *home_automation_product_line* is constructed according to Construction 2. In this case, we have

$$\mathrm{R}_{\mathrm{home\_automation\_product\_line}} = \mathrm{D}_{\mathrm{home\_automation\_product\_line}}.$$

We construct the referencing label set of the aspect of Case $1(a)$ using Table 1, which is the same as its defining label set. We have

$$\mathrm{R}_{\mathrm{home\_automation\_product\_line}} \cup \mathrm{R}_{\mathrm{case1a\_aspect}} = \mathrm{D}_{\mathrm{home\_automation\_product\_line}} \cup \mathrm{D}_{\mathrm{case1a\_aspect}}.$$

Based on Definition 10, we verified that the considered aspect is reference-valid w.r.t. its base specification.

As a further example, we make a minor modification to *Daw_dev* in the original Specification *home_automation_product_line* as shown in Figure 16. The new specification is called Specification *home_automation_two*. With this modification, the defining label set of Specification *home_automation_two* remains the same as $\mathrm{D}_{\mathrm{home\_automation\_product\_line}}$, while the referencing label set becomes:

$$\mathrm{R}_{\mathrm{home\_automation\_two}} = \mathrm{R}_{\mathrm{home\_automation\_product\_line}} \cup \{\mathit{fin\_reader}\}$$

According to Definition 6, the modified specification is reference-invalid, while according to Definition 10, the aspect of Case $1(a)$ is reference-valid w.r.t. the Specification *home_auto*–

mation_two. In other words, the specification becomes reference-valid after weaving[2] the considered aspect. The above example illustrates how the proposed technique can be used to ensure safe incrementation of systems with regard to reference-validity, even if the original specification is reference-invalid.

Specification *home_automation_two*:

...

Daw_dev = tim_con· (1+wea_sen) · (1+fin_reader)

...

Figure 16: Modified specification of Specification *home_automation_product_line*

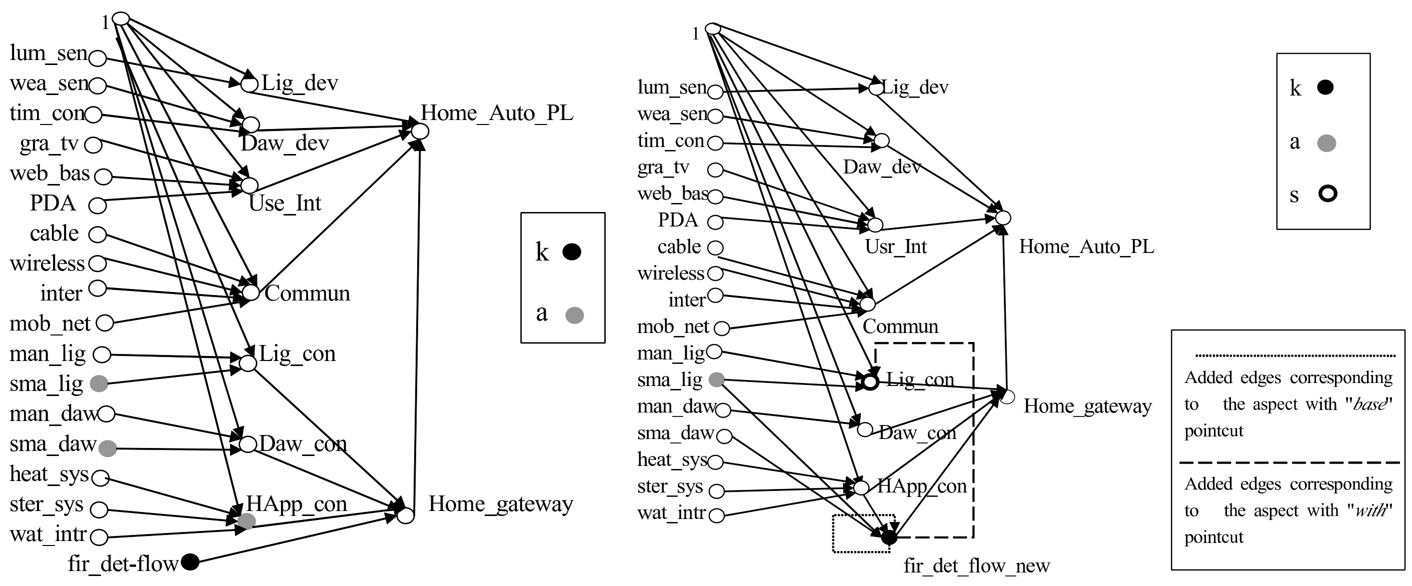
Verification of dependency-validity We apply the propositions given in Section 4.1.2 to verify the dependency-validity criteria of an aspect w.r.t. a given base specification. Take the several aspects given in the Case 2 of our case study as examples.

Based on Table 1, the referencing label set of the aspect of Case 2(a) is {*fir_det_flow*}, whose intersection with $\mathrm{D}_{\mathrm{home\_automation}}$ is empty. According to Prop. 2, we can conclude directly that, even without constructing the dependency digraph of the base specification, the aspect of Case 2(a) is dependency-valid w.r.t. its base specification. Assume Specification *home_automation_three* is the resulting specification after weaving the aspect of Case 2(a). We construct the dependency digraph of Specification *home_automation_three* in Figure 17(a) according to Construction 3. According to Definition 8, the resulting specification is dependency-valid, which confirms that the aspect is dependency-valid w.r.t. its base specification.

Consequently, the base specification for Case 2(b) is Specification *home_automation_three*. In Figure 17(a), gray vertices represent vertices in both the referencing label set of the aspect of Case 2(b) and the defining label set of the base specification, and the black vertex corresponds to k denoted according to Construction 5. Since there is no path from k to any vertex corresponding to *sma_lig*, *sma_daw* and *HApp_con*, Prop. 2 indicates that the above aspect is dependency-valid w.r.t. its corresponding base specification. Assuming the resulting specification after weaving the aspect of Case 2(b) is called Specification *home_automation_four*. We construct the dependency digraph of Specification *home_automation_four* in Figure 17(b), which is loop-free and cycle-free.

We next consider an example illustrating a case where the aspect is dependency-invalid w.r.t. its base specification. We use an *inclusion* pointcut to capture join points where *sma_lig* is being referenced and specify an aspect as follows:

[2]We assume that the model of product family algebra is set-based, which means the operator · is idempotent; we do not allow duplication of features.



(a) Corresponding to Specification *home_automation_three*

(b) Corresponding to Specification *home_automation_four*

Figure 17: The dependency digraphs of PFA Specifications

$$\begin{aligned} \text{Aspect} \quad & \text{jp} = \text{jp} \cdot \text{fir\_det\_flow\_new} \\ \text{where} \quad & \text{jp} \in \Big(\text{base, true, } \mathit{inclusion}(\text{sma\_lig})\Big) \end{aligned}$$

Take the specification corresponding to Figure 17(b) as the base specification. With regard to the above aspect, we denote vertices in Figure 17(b) in the same way as in Figure 17(a). According to the first item in Prop. 9, the aspect is dependency-invalid w.r.t. its base specification; there is a path from the vertex of *sma_lig* to the vertex of *fir_det_flow_new*. However, assume we change the scope of the above pointcut to be *within*(*Lig_con*) instead of *base*. Based on Construction 3, the vertex in Figure 17(b) represented by a bold circle corresponds to vertex *s*. Due to the second item in Prop. 9, the modified aspect is dependency-valid w.r.t. its base specification. In Figure 17(b), the dashed edges represent new edges that will be introduced by weaving the two aspects that have pointcuts with scope *base* and *within*, respectively. The dashed edges illustrate that the first aspect will introduce a loop in the dependency digraph, while the second aspect will not.

5 Conclusion and Future Work

In this technical report, we have proposed an aspect-oriented specification language, called AO-PFA that extends aspect-oriented notations to product family algebra specifications. The AO-PFA language provides full facilities for formally articulating aspects and base systems at the feature-modeling level. Moreover, we presented a formal verification technique of aspectual composition in the context of AO-PFA. We defined a set of validity criteria for PFA base specifications and aspectual composition. The proposed approach enables the detection of definition, reference, and dependency invalid aspects.

We are using the work presented in [18] as the basis for our ongoing work on introducing finer granularity aspects (at the state level rather than the feature level). The objective from 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. [18] demonstrated 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 the weaving aspects should lead to, among others, the specification of a product given using a slight variation of Dijkstra's guarded command [10]. Consequently, the work presented in this paper aims at detecting interferences caused by aspectual compositions at the syntactic level. In other words, features are taken as "black-boxes". To detect interferences caused by aspectual composition at the semantic level, we consider the work presented in [18] as the basis for our ongoing work on introducing finer granularity features.

On the other hand from the application perspective, we aim to apply our technique to larger scale applications. Basically, we intend to focus on applying our approach to handle security issues in product families of software systems. One major difficulty related to security concerns is caused by its crosscutting characteristic. As an emerging technique for handling crosscutting concerns, the aspect-oriented paradigm is naturally recognized as a promising technique for security. However, most current aspect-oriented techniques are concentrating on the implementation and detailed design level [28]. Therefore, applying the proposed method to security would contribute to the early stages of security engineering, which interest both the security and aspect-oriented development communities.

References

[1] Mauricio Alférez, João Santos, Ana Moreira, Alessandro Garcia, Uirá Kulesza, João Araújo, and Vasco Amaral. Multi-view composition language for software product line requirements. In *Proc. of the 2nd International Conference on Software Language Engineering*, 2009.

[2] Fadil Alturki. Jory: A tool for feature modelling based on product families algebra and bdds. M.a.sc. thesis, McMaster University, Hamilton, Ontario, Canada, March 2010.

[3] Fadil Alturki and Ridha Khedri. A tool for formal feature modeling based on bdds and product families algebra. In *13th Workshop on Requirement Engineering*, 2010.

[4] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: Aspects and features in concert. In *Proc. of the International Conference on Software Engineering*, 2006.

[5] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(2010):1022–1047, 2010.

[6] J. Araújo and A. Moreira. An Aspectual Use Case Driven Approach. In *VIII Jornadas de Ingeniera de Software Bases de Datos*, 2003.

[7] D. Balzarotti and M. Monga. Using program slicing to anlyze aspect-oriented composition. In *Proc. of Foundations of Aspect Language (FOAL)*, 2004.

[8] Ruzanna Chitchyan, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke, and Andrew Jackson. Survey of analysis and design approach. Survey, AOSD-Europe, 2005.

[9] Krzysztof Czarnecki. *Generative Programming, Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, October 1998.

[10] E. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[11] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS approach-domain modeling with features, use cases and use realization. In *Proc. of 9th International Conference on Software Product Lines*, 2005.

[12] Rober E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on advanced separation of concerns, Research Institute for Advanced Computer Science, May 2001.

[13] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating features modeling with the RSEB. In *Proc. of the 5th International Conference on Software Reuse*, 1998.

[14] Iris Groher and Markus Voelter. Xweave: Models and aspects in concert. In *Proc. of the 10th Workshop on Aspect-Oriented Modelling*, 2007.

[15] Peter Höfner, Ridha Khedri, and Bernhard Möller. Feature algebra. In J. Misra, T. Nipknow, and E. Sekerinski, editors, *Formal Methods, Lecture Notes in Computer Science*, volume 4085, pages 300–315. Springer-Verlag, 2006.

[16] Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. In *Proc. of 6th IEEE International Conference on Software Engineering and Formal Methods*, 2008.

[17] Peter Höfner, Ridha Khedri, and Bernhard Möller. An algebra of product families. *Software and Systems Modeling*, 10(2):161–182, 2011.

[18] Peter Höfner, Ridha Khedri, and Bernhard Möller. Supplementing product families with behaviour. *International Journal of Informatics*, pages 245 – 266, 2011. Special Issue II: Foundations and Practice of Systems and Software Engineering, Festschrift in Honor of Manfred Broy.

[19] I. Jacobson and P.W. Ng. *Aspect-Oriented Software Development with Use Cases.* Addison Wesey Professional, December 2004.

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

[21] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 2001.

[22] Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[23] S. Katz and M. Sihman. Aspect validation using model checking. In *Proc. of the International Symposium on Verification in Honor of Zohar Manna*, 2003.

[24] Shmuel Katz. A Survey of Verification and Static Analysis for Aspects. Technical report, AOSD Europe, 2005.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[26] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. of the International Conference on Foundations of Software Engineering*, 2004.

[27] Neil Loughran and Awais Rashid. Framed aspect: Support variability and configurability for AOP. In *Proc. of ICSR*, 2004.

[28] Neil Loughran, Awais Rashid, Ruzanna Chitchyan, and et al.. A domain analysis of key concerns - known and new candidates. Research, AOSD-Europe, 2006.

[29] Neil Loughran, Américo Sampaion, and Awais Rashid. From requirements documents to feature model for aspect oriented product line implementation. In *MDD in Product Line at MODELS*, 2005.

[30] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proc. of the 12th ACM International Symposium on Foundations of Software Engineering*, 2004.

[31] Klaus Pohl. *Software product line engineering: foundations, principles, and techniques*, chapter 3. Springer-Verlag, Berlin, Heidelberg, 2005.

[32] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and Composition of Aspectual Requirements. In *Proceedings of the 2nd International Conference on Aspect Oriented Software Development*, pages 11–20, 2003.

[33] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicites. In *Proc. of 6th Conference on Integrated and Design Process Technology*, 2002.

[34] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. of 12th Symposium on the Foundations of Software Engineering*, 2004.

[35] Frans Sanen, Ruzanna Chitchyan, and Lodewijk Bergmans. Aspect, Dependencies and Interactions Report on the Workshop ADI at ECOOP 2007. In *Proc. of the 2007 Conference on Object-Oriented Technology*, 2008.

[36] D. Sereni and O. de Moor. Static analysis of aspects. In *Proc. of Aspect-Oriented Software Development (AOSD)*, 2003.

[37] M. Storzer and J. Krinke. Interference analysis for AspectJ. In *Proc. of Foundations of Aspect Language (FOAL)*, 2003.

[38] Detlef Streiferdt. *Family-Oriented Requirements Engineering.* PhD thesis, Technical University Ilmenau, IImenau, Germany, 2004.

[39] J. van Gurp, J. Bosch, and M. Svahnberg. On the notation of variability in software product lines. In *Proc. of the Working IEEE/IFIP Conference on Software Architecture*, page 45, 2001.

[40] D.M. Weiss and C.T.R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison Wesley Longman, Inc., New York, 1999.

[41] J. Whitter and J. Araújo. Scenario modelling with aspects. In *IEE Proceedings of Software Special Issue*, 2004.