ASPECT-ORIENTED PRODUCT FAMILY MODELING

ASPECT-ORIENTED PRODUCT FAMILY MODELING

BY

QINGLEI ZHANG, B.Eng., M.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Doctor of Philosophy

Doctor of Philosophy (2013) McMaster University

(Computing & Software) Hamilton, Ontario, Canada

TITLE:	Aspect-oriented Product Family Modeling
AUTHOR:	Qinglei Zhang M.Sc. (Trent University) B.Eng. (University of Science and Technology of China)
SUPERVISOR:	Dr. Ridha Khedri
NUMBER OF PAGES:	xviii, 216

To my family

Abstract

Practice experience of different organizations has revealed that it is advantageous to develop a set of related products from core assets instead of developing them one by one independently. The set of related products is referred to as a product family, and feature-modeling is a widely used technique to capture the commonalities and variabilities of a product family in terms of "features". With the growing complexity of software product families in several software industries, the development, maintenance and evolution of complex and large feature models are among the main challenges faced by feature-modeling practitioners. In particular, more sophisticated feature modeling techniques are required to address the problems caused by unanticipated changes and crosscutting concerns in feature models.

This thesis tackles the above challenges in feature-modeling by adopting the aspect-oriented paradigm at the feature-modeling level. I firstly introduce a specification language, called AO-PFA, which is an extension of the Product Family Algebra (PFA) language. I then proposed a formal verification technique to check the compatibility of aspects with their base specifications in AO-PFA. In the aspect-oriented paradigm, the process of combining aspects with base specifications is referred to as the weaving process. I finally discussed how to perform the weaving process in AO-PFA.

By proposing a systematic approach to extend product family algebra with the abilities of specifying, verifying, and weaving aspects, we are able to handle the difficulties that arise from crosscutting concerns and unanticipated changes in large-scale feature models. An original feature model is considered as a base specification, while the changes to the feature model are specified as aspects. The formal verification and weaving techniques ensure that the changes are correctly and properly propagated to the original feature model.

Acknowledgements

I am grateful to the McMaster University for giving me the opportunity and financial support to pursue my PhD degree. I would like to thank all who have helped me throughout the completion of my doctoral journey.

I would first and foremost like to express my sincere gratitude to my supervisor, Dr. Ridha Khedri, for his substantial support and encouragement throughout the process of my research. His expertise, guidance and motivation were the fundamental to make this thesis possible.

I would like to thank my Supervisor Committee Members: Dr. William Farmer and Dr. Ryszard Janicki for following up on my research and giving useful comments. Also, I would like to extend my thanks to Dr. Wahab Hamou-Lhadj for accepting to be the external examiner in my committee and giving essential comments that added values to my work. In addition, I would like to thank Mr. Jason Jaskolka for his valuable discussions and suggestions.

Last but not the least, a very special thank you goes to my parents for all their love, encouragement, and support.

Contents

List of Tables

List of Figures

List of Symbols

A_{def}	Set of new labels induced by the aspect	148
B_{def}	Set of labels in the base specification which are defined by the selected join points	99
$\mathcal{C}$	Set of all potential reference join points associated with constrains	116
$Cp(S)$	Set of constrains specified by a specification S	113
$\preceq$	Subterm relation	115
$\mathcal{DJ}$	Set of all potential definition join points	116
D_{S}	Set representing definition labels in the specification S	92
$\mathcal{E}$	Set of all potential reference join points associated with labeled family equations	116
E_add_{A}	Set of edges added to a label dependency digraph by composing an aspect A	95
E_del_{A}	Set of edges deleted from a label dependency digraph by composing an aspect A	95
E_f	Equational theory related to axioms of a commutative idempotent semiring	112

$Eq(S)$	Set of equations specified by a specification S	113
E_{spec}	Equational theory related to the base specification	131
E_tk	Set of equations decided by the kind pointcut	120
$E\_add(p)$	Set of equations decided by the scope pointcut at a potential join point p	121
$\mathcal{F}$	Signature of a commutative idempotent semiring	112
G_{S}	Digraph representing label dependencies in the specification S	93
JP	Set of all potential join points	116
$\mathfrak{L}(S)$	Set of labels specified by a specification S	113
M_{S}	Multi-set of definition labels specified in the specification S	92
$\mathcal{M}$	Signature of a monoid	113
$\mathcal{RJ}$	Set of all potential reference join points	116
$R(E_f)$	Set of rewriting rules derived from E_f	131
$R(E_{spec})$	Set of rewriting rules derived from E_{spec}	132
R_{S}	Set representing reference labels in the specification S	93
Th_{pfa}	Equation theory associated with the weaving process in AO-PFA	120
V	Set of variables w.r.t. the signature of $\mathcal{F}$	113
$Walk(u, v)$	List of vertices along a walk from u to v in a digraph	98
Γ	Alphabet of the PFA language	113
a	Product family algebra term specified by the advice	101

k	Product family algebra term specified by the kind pointcut	98
s	Product family algebra term specified by the scope pointcut	98
tk	Type of the kind pointcut	120
ts	Type of the scope pointcut	102
(u, v)–path	Path starting with vertex u and ending with vertex v	52

List of Abbreviations

ADI	Aspectual Dependencies and Interaction
AOAD	Aspect-Oriented Architecture Design
AODD	Aspect-Oriented Detailed Design
AOP	Aspect-Oriented Programming
AORE	Aspect-Oriented Requirement Engineering
AOSD	Aspect-Oriented Software Development
AP	Adaptive Programming
CF	Composition Filter
FOSD	Feature-Oriented Software Development
MDSOC	Multi-Dimensional Separation of Concerns
PFE	Product Family Engineering
SOP	Subjective-Oriented Programming
TRS	Term Rewriting System

Chapter 1

Introduction

1.1 General Background and Motivation

Practice experience of different organizations has revealed that it is advantageous to develop a set of related products from core assets instead of developing them one by one independently [CN02]. Such a set of related products, which usually together address a particular market segment or mission, is referred to as a *product family*. Product Family Engineering (PFE) is an emerging software development discipline proposing structure processes and techniques to facilitate the development of product families. Rather than developing product families from scratch, or in an arbitrary fashion, product family engineering aims to be a systematic, organized and effective approach for software production [Coh02]. In particular, product family engineering can improve productivity, increase software quality, reduce cost and labour needs, and decrease the time to market [MNJP02]. One of the key issues in product family engineering is to capture the commonality and variability occurring in a product family. There are in general two main phases

for product family engineering: domain engineering and application engineering. Domain engineering explores all variable and common requirements in the target product market. It also establishes reusable core assets from a domain perspective. Application engineering specifies the requirements of individual products, and then a product is constructed by selecting and configuring the specific parts from the core assets.

Feature-Oriented Software Development (FOSD) is one of the widely used methodologies in product family engineering, where the commonalities and variabilities are captured in terms of features. The term *feature* has slightly different meanings in the literature. We adopt the definition used in [LKL02], where any "visible prominent and distinctive characteristic relevant to certain stakeholders (e.g., users, analysts, designers and developers)" is recognized as a feature. There are four stages of the feature-oriented software development process: (1) domain analysis, (2) domain design/specification, (3) domain implementation, and (4) product configuration and generation. Domain analysis aims to identify which features are part of the domain and how they are related. The essential structural and behavioural properties of the involved features are specified at the domain design stage, and a set of first-class artifacts are created for each feature at the domain implementation stage. Following the feature selection of an individual product, the product is generated automatically at the last stage. A key ultimate goal of the feature-oriented software development is to achieve the automatic derivation of products. Figure 1.1 illustrates the state of the art of the feature-oriented software development with respect to the different development stages.

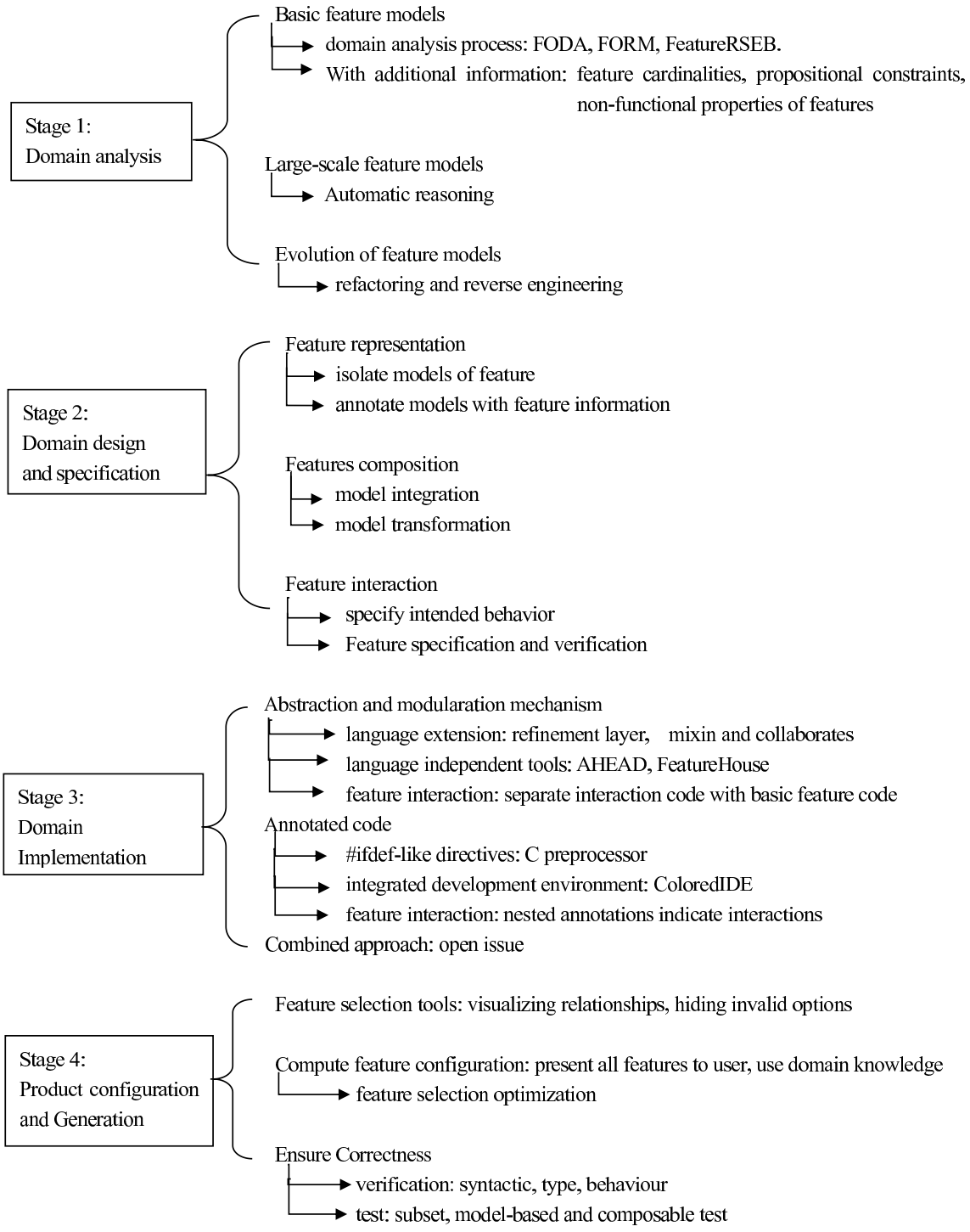

Figure 1.1: The state of the art of feature-oriented software development

1.1.1 Feature-Modeling

In this thesis, I focus on the topic of *feature-modeling*, by which the commonalities and variabilities of product families are specified and managed with feature models. In particular, a *feature model* describes a set of valid combination of features such that each combination corresponds to a member in a product family. Since being firstly introduced and used by Kang et al. in 1990 [KCH$^+$90], feature-modeling techniques have been well accepted and applied in both academic and industrial projects. Feature-modeling is the central activity at the domain analysis stage of the feature-oriented software development. Moreover, to achieve the full potential of feature-oriented software development, feature models are used as the essential basis to guide the following processes of domain design, implementation, and product configuration and generation.

With the increasing complexity of software product families in practice, it is common to find feature models involving hundreds or even thousands of features. A large feature model cannot be understood and analyzed if they are treated as a monolithic entity with traditional feature-modeling techniques. The development, maintenance, and evolution of complex and large feature models are among the main challenges faced by feature-modeling practitioners. In particular, it is hard to handle unanticipated changes in large feature models. To illustrate this point, we use a security related situation. An authentication feature is in charge of identifying the caller agent that remotely communicates with a sub-system in a product family. After a while, we find that the feature interaction of the authentication feature with other features enable an intruder to take control of the system. The remedy of the detected defect in the product family leads to the introduction of new variability in

the family or to the amendment of the existing variability by confining it to some products but not others. This evolution scenario due to the security issue may not be anticipated at the time of the feature-modeling stage. Consequently, it can be a tedious and costly work to recognize and make such unanticipated changes in feature models. The difficulties increase if the unanticipated changes happen when a serious defect is detected after the product family has been put on the market. The question then becomes how to supersede the current feature model of a family by a new one in an effective and efficient way.

More sophisticated feature-modeling techniques are required to address the above problems in modeling large-scale feature models. One well known and effective way to manage the complexity of large systems is through the use of the separation of concerns, which was initially coincided with David Parnas' idea of criteria for decomposing systems [Par72]. Recent research on the composition of feature models for handling large scale product families is related to the idea of adapting the principle of separation of concerns at the feature-modeling level. In other words, a large feature model is developed by composing multiple feature models such that each feature model corresponds to the modularization of a particular concern. However, current modularization approaches at feature-modeling level still have limitation for handling *crosscutting concerns*. Crosscutting concerns inherently scatter over and intertwine with other concerns. Without a proper modularization mechanism, a crosscutting concern cannot be reused but has to be implemented whenever it is required. Also, when a concern is implemented everywhere over the whole system and intertwined with other concerns, the maintainer would need to manually inspect and update the models at all locations that relate

to this concern. The crosscutting concern is recognized as a major contributor that impedes the development, maintenance, and evolution of complex systems [HL95]. In the next subsection, I discuss the crosscutting concerns and problems caused by them.

1.1.2 Aspect-Orientated Paradigm

In response to the challenges of large feature models as mentioned in the previous section, we adopt the *aspect-oriented paradigm* at the feature-modeling level to support the modularization of feature models. In this section, we introduce the aspect-oriented paradigm and its usage in general. Then, in the next section, we discuss its specific application to the development of product families.

On the motivation for the aspect-oriented paradigm: crosscutting concerns

Traditional high level programming languages were originally intended to support the module abstraction kind of separation of concerns. Program designers only can decompose the system with respect to one criterion [EFB01]. For instance, object-oriented programming designers decompose a system as a collection of interacting objects. Process-oriented programming designers decompose a system as a set of parallel processes with shared data structures. Those concerns are called *base concerns* or *core concerns* [HL95], because they usually capture the most essential functionalities of the application. However, not all required computations can be neatly separated and encapsulated as base concerns. Computations that are not related to base concerns are addressed in the code whenever necessary. In

other words, the implementation of non-core concerns may be spread over multiple modules or intertwined with other concerns, or both [HL95]. Since those scattering and tangling concerns crosscut the base concerns, they are called *crosscutting concerns*.

To further explain the nature of crosscutting concerns and base concerns, we use the example of a simple *figure editor* system, which is borrowed from [EAK+1a]. Figure 1.2 depicts a simple figure editor, where the system is decomposed following an object-oriented style and each class encapsulates a graphical element. Such a decomposition ensures that each class is closely related (cohesion) and the interactions between classes are well-defined (coupling). Thus the representations of graphic elements (e.g.,*Point* and *Line* in the example) are considered as base concerns. Besides base concerns, there are other concerns such as *Display Update*. *Display Update* requires every method that moves a figure element to notify the screen manager about the movement. The method labels written in italic font in Figure 1.2 indicate all methods that should implement such notifications for the *Display Update* concern. It is obvious that the box for *Display Update* cuts across the boxes for *Point* and *Line*. Hence, the *Display Update* concern is referred to as the crosscutting concern. More examples of crosscutting concerns for different applications regarding different programming languages are given in [KLM+97].

Although a crosscutting concern is related to a particular dominant decomposition for base concerns, some issues are identified as crosscutting concerns in

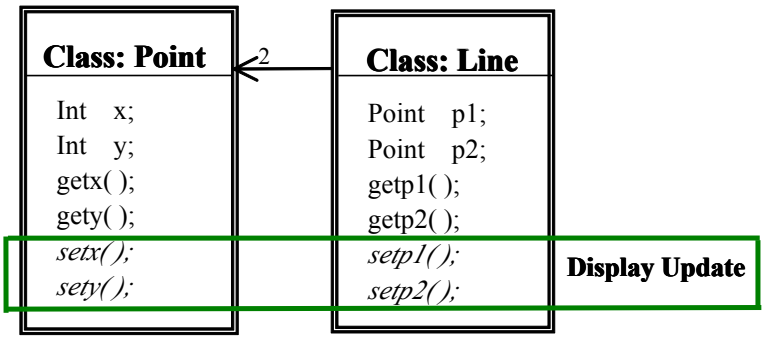

Figure 1.2: An editor example to illustrate base and crosscutting concerns

a common sense. Crosscutting concerns are frequently recognised in many applications and usually used as supporting roles in those applications, such as information security, memory management, error detection, or logging and monitoring [KLM+97]. Furthermore, special crosscutting concerns are also identified in certain application domains, such as synchronization in parallel applications, location control in distribution computations, real time constrains in the real time systems, failure recovery in fault-tolerance systems [HL95], and product features for product family engineering [LKF02]. In particular, the implementation of one feature may be scattered over several basic abstractions, or more than one feature may be implemented within one basic abstraction. *Feature-oriented programming* is primarily developed to implement products through the composition of features. However, while many features can be developed independently, some features inherently have dependencies with others and still cannot be implemented elegantly using feature-oriented programming.

Handling crosscutting concerns is an important issue for developing, testing and verifying the software. Implementing a crosscutting concern requires the cooperation of several software developers as the concern cannot be assigned to a

single development team. Testing a crosscutting concern needs knowledge about the modules affected by the concern. Representation of and reasoning on a crosscutting concern make formal verification much harder as the dependencies of a crosscutting concern with other modules are very often implicit.

On the basis of the aspect-oriented paradigm: terms and usages

To reduce difficulties in handling crosscutting concerns, it is helpful to explicitly modularize those crosscutting concerns. The *aspect* concept was first introduced in the language AspectJ [KLM+97] by Gregor Kiczales et al. in 1997. AspectJ is an extension for the Java programming language to explicitly encapsulate and implement every crosscutting concern in one module. Gregor Kiczales et al. defined an aspect as a module to implement the "property that cannot be cleanly encapsulated in a generalized procedure". Moreover, a component is defined as a module to implement the "system functionality that can be cleanly encapsulated in a generalized procedure" [KLM+97]. Besides the concept of aspect, some other important concepts are defined in AspectJ as follows:

- *Join point*: It defines the point at the execution of the base program where the aspect could be introduced.
- *Pointcut*: It selects a set of join points where a certain aspect should be introduced.
- *Advice*: It defines the behaviours which should be introduced to the selected join points.

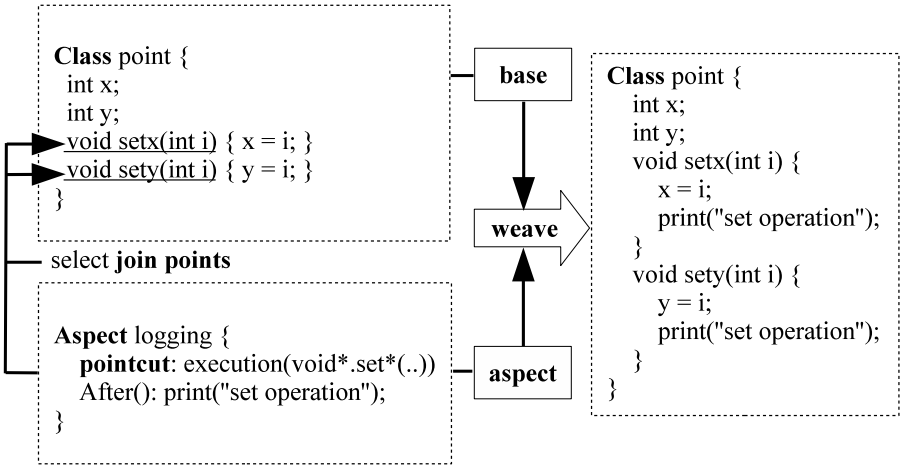

Figure 1.3: Example of a logging aspect

- *Weave*: It is the process of combining the aspects with the rest of the programs.

Figure 1.3 shows a simple example written in AspectJ. The aspect logging is composed of the pointcut and the advice. The pointcut selects two join points (underlined instructions) in the base code. After weaving, those selected join points are advised by the aspect as shown in the right rectangle.

With the AspectJ language, a system is composed of a set of aspects and a set of components. In general, aspects are used to modularize crosscutting concerns, while components are used to modularize base concerns. Such a decomposition paradigm for the development of systems is referred to as the aspect-oriented paradigm. All those concepts mentioned above that are introduced by AspectJ are commonly used for all aspect-oriented techniques. The aspect-oriented paradigm aims to improve software qualities, such as reusability, maintainability, adaptability and extensibility, by properly handling crosscutting concerns. In particular, the aspect-oriented paradigm provides the explicit abstraction and the non-invasive

composition mechanism for crosscutting concerns. The advice defines the behaviour which implements a crosscutting concern, while the pointcut identifies join points where the crosscutting concern should be introduced. The reusability and maintainability of the software are improved by encapsulating crosscutting concerns into a single module.

On the development of the aspect-oriented paradigm: early aspects and AOSD

The aspect-oriented paradigm not only relates to the implementation of crosscutting concerns, but also relates to the analysis and design of the crosscutting concerns. For example, aspects derived from the low implementation level could be cached and buffered [EAK+1a]. Aspects derived from analysis and design levels could be a non-functional requirement such as quality of service or a functional requirement such as business rules. Aspects at those analysis and design levels are sometimes referred to as *early aspects*. Early aspects can help the understanding and identification of crosscutting concerns at an early stage, and enable reaching an early trade-off between concerns. Research shows that if aspects are employed with non-crosscutting concerns at the implementation level, it may increase the complexity of the software, or even decrease the maintainability of the software. Early aspects, on the other hand, are not necessarily mapped as aspects at the implementation level. For example, the aspects identified in the requirement level, could either be architectural components or design decisions during the following development stages. A more comprehensive discussion on the main benefits of dealing with early aspects is given in [batmaccRS+05, RM06].

All techniques and tools developed to support aspects in the whole life-cycle of software development are referred to as *Aspect-Oriented Software Development (AOSD)*. In particular, those techniques are *Aspect-Oriented Programming (AOP)* techniques [KLM+97, BA01, OT01, LOO01], *Aspect-Oriented Requirement Engineering (AORE)* techniques [AM03, RMA03, YLM04, WA04, JN04, MAR05, MRA05], *Aspect-Oriented Architecture Design (AOAD)* techniques [PFT03, Kan03, BCD+04, KGdLvS04, Tek04], and *Aspect-Oriented Detailed Design (AODD)* techniques [Her02, SHU02, AEB03, BC04]. Due to the difference in the granularity of concern abstractions at different stages, the contents of aspects vary slightly at different stages. At the requirement stage, aspects refer to crosscutting properties that have broad scopes of effect on the other requirements or architectural components. At the architecture design stage, aspects focus on concerns which crosscut several architecture components that are factored using conventional architecture approaches. At the detailed design stage, aspects relate to behaviours and structures that are unavoidably scattered and tangled in the detailed design documents [batmaccRS+05]. At the programming stages, aspects encapsulate code that crosscuts many modules that are defined by programming languages. An extensive survey on those techniques for aspect-oriented software development is presented in Section 2.1. The survey indicates that more mature approaches are still under development and many challenges remain in the area of aspect-oriented software development. Besides approaches at each development stage, preserving aspects along different development stages is also an open issue for aspect-oriented software development [batmaccRS+05]. Moreover, much more effort is required to extract the common characteristic of all aspect-oriented techniques. Currently,

only several initial reference models [BMN+06, SSK+07] have been proposed for the identification and analysis of essential elements of the aspect-oriented paradigm. Moreover, mathematical models which are constructed to capture the mechanism of the aspect-oriented paradigm are still quite few in the literature of aspect-oriented software development [MK03].

1.1.3 Feature-Modeling with the Aspect-Oriented Paradigm

In this section, we discuss the benefits of adapting the aspect-oriented paradigm at the feature-modeling level. With regard to the problems mentioned in Section 1.1.1 for large-scale feature models, we particularly show how to tackle unanticipated changes and crosscutting concerns in feature models by using the aspect-oriented paradigm.

Handling unanticipated changes in feature models

The aspect-oriented paradigm can be taken as a candidate approach for modularizing large feature models, which can improve the maintainability, extensibility, reusability, and adaptability of feature models [EFB01]. More specially, Filman and Friedman [FF01] characterize the aspect-oriented paradigm as *quantification* and *obliviousness*. Quantification means the ability to make quantified statements over the underlying code, while obliviousness indicates that the original code need not be prepared for being extended with new aspects. These two characteristics of the aspect-oriented paradigm provide a way to propagate unanticipated changes in feature models by composing aspects. The composition mechanism of aspects

enables generic changing operations on feature models: 1) a set of feature combinations can be introduced to a feature model at one or multiple place(s), and 2) a set of feature combinations can be removed from a feature model at one or multiple place(s). With the aspect-oriented paradigm, the pointcut is responsible for identifying where to make changes in the original feature models, while the advice can capture a set of features in arbitrary combinations. Moreover, since join points can be skipped in an advice, using such aspects provides a way to remove features at selected places from the feature models.

Handling crosscutting concerns in feature models

Besides improving the modifiability of large feature models, adapting the aspect-oriented paradigm at the feature-modeling level also aims to achieve a systematic aspect-oriented development for product families. We have mentioned earlier that some features can be considered as examples of crosscutting concerns (See page 8). Moreover, to moderate the complexity for handling crosscutting concerns at the implementation level, it is advantageous to trace crosscutting concerns from the early analysis and design stages. Therefore, adopting the aspect-oriented paradigm at the feature-modeling level is one of the essential steps to properly handle crosscutting concerns throughout the whole life-cycle of product family engineering.

Features that can not be mapped to a singleton basic abstraction are crosscutting features. The implementation of crosscutting features bears the same problems, such as maintenance and evolution, as general crosscutting concerns. In the literature [Gri00a], there are three kinds of strategies to handle those crosscutting

features. One strategy is to trace those crosscutting features by tools from requirements to their design and coding. However, the tracing strategy could be hard to control when the system becomes extremely complex and the features crosscut a lot of other features. Another strategy is to introduce a new layer to encapsulate crosscutting features and their dependent features. This strategy has the drawback that crosscutting features have to be introduced at every required point within the base program [MO04]. The third strategy is to separate those crosscutting features and then to compose them when necessary [CRB04]. This strategy can overcome the weakness of the other two strategies. The aspect-oriented paradigm is an illustration of the third strategy. In particular, aspects can encapsulate crosscutting features as well as all locations at where to introduce them. In [Gri00b], Griss summarized the advantages of the composition and weaving approaches for dealing with crosscutting concerns in product families.

1.2 Problem Statement, Objectives, and Methodology

This thesis focuses on developing a systematic formal approach that adapts the aspect-oriented paradigm to feature-modeling. In particular, we propose an aspect-oriented specification language for feature-modeling. Feature model specifications are considered as bridges that narrow the gap between the problem domain and the solution domain. Moreover, tool assistance is demanded for the management and analysis of large feature models. A formal specification language for feature-modeling is the prerequisite that enables tools to conduct complex activities on

feature models.

In this thesis, we aim to complement the current product family algebra approach with the ability to specify feature models using the aspect-oriented paradigm. As we have discussed, developing a systematic way for composing feature models is a primary issue to manage the complexity of large feature models. But only a few approaches have been proposed with regard to this issue [ACLF10]. The current product family algebra technique [HKM08] proposes one of those attempts that integrates different concerns of feature models by *view reconciliation*. Each feature model partially describes commonalities and variabilities of the considered view for the product family, and constraints are defined to exclude all products that are generated from incompatible view integration. View reconciliation is an approach that adapts the classic multi-view integration approach to feature-modeling techniques. However, this approach has bottlenecks for separating and composing crosscutting concerns as the classic multi-view integration approach [batmaccRS$^+$05]. For instance, a security flaw detected in a subfamily or even a product could require the removal of a feature from the original products. However, constraints only allow one to remove products from the original product families. We are unable to change the feature model properly with the current product family algebra approach. Moreover, with only view reconciliation, it can be a quite tedious work to integrate a crosscutting concern in a feature model since various constraints can be required for the composition of only a simple crosscutting concern. In summary, adapting the aspect-oriented paradigm to product family algebra helps to improve the extensibility and modifiability of the current view reconciliation approach of product family algebra.

The *modularization* and *composition* of aspects are two of the essential issues that need to be handled to achieve the research goal of adapting the aspect-oriented paradigm to product family algebra. In particular, we set the following three objectives.

The first objective is to specify aspects and base specifications at the feature-modeling level. Currently, PFA specifications are used to specify feature models based on product family algebra. We extent the existing PFA specification language with the aspect-oriented paradigm. The PFA specifications are considered as the base specifications, and we propose a specification language for aspect specifications. The primary issue for specifying aspects is to identify those essential terms of the aspect-oriented paradigm that are relevant in the context of product family algebra. Generically, join points and the advice are specified in terms of product family terms, while a pointcut aims to capture certain attributes of those product family terms. We analyze the exact form of join points, the advice and the pointcut to enable the generic changing on PFA specifications.

The second objective is to verify aspectual composition at the feature-modeling level. As a new emerging paradigm to deal with product families, adopting the aspect-oriented paradigm in an effective and safe way is still a topic under discussion. Aspect-oriented approaches are reasonable only if the complexity of aspectual composition can be well handled. The verification of aspectual compositions is much more complicated since crosscutting concerns inherently have broad impacts on many other concerns. Research related to *Aspectual Dependencies and Interaction (ADI)* is a common interest in the literature regarding appropriate

composition of aspects. The main topics of ADI are about identifying, detecting and handling aspectual dependencies and interactions. Some efforts are also taken on the classification of those dependencies and interactions. In our context, we analyze the aspectual dependencies and interaction problems associated to the specification language proposed in the previous objective. We intend to identify different composition patterns of aspects. Firstly, we categorize different aspects according to their direct effects at the selected join points. Secondly, we analyze the indirect effects of aspects on the base systems. To avoid unintended aspectual dependencies and interaction, we detect all potential scenarios that can spoil a system when the aspectual composition is performed.

The third objective is to automate the weaving of aspect and base specifications at the feature-modeling level. The general weaving processes for aspect-oriented languages can be decomposed into several activities: *join point evaluation*, *pointcut matching*, *advice binding*, and *weaving execution*. We formalize the semantics of each activity in the context of the proposed specification language. A key problem of the weaving process in the context of product family algebra is extracting subterms from product family terms in the base specification, and then replacing each extracted subterm (selected join points) by another term (advice). This problem is related to what is known as the *word problem*, which is undecidable in general. It is necessary to provide a decidable procedure for the word problem introduced in our context to automate the weaving process. We use the technique of term rewriting systems to solve the word problem.

1.3 Contributions

Through the fulfillment of the above objectives, the research has produced the following contributions:

(1) The design of AO-PFA (Section 4.2): I propose a specification language, AO-PFA that extends the aspect-oriented paradigm to product family algebra. The AO-PFA language provides full facilities for articulating aspects, advice, and pointcuts in feature models. I also illustrate the scope and flexibility of the proposed language through the discussion of several feature-modeling situations. Results regarding this topic have been published in [ZKJ12a, ZKJ13, ZKJ11].

(2) The classification of aspects in AO-PFA (Section 4.3): I present a classification system for aspects in the context of AO-PFA. In particular, we distinguish aspects in accordance to their augmenting, narrowing, and replacing effects upon join points. The categories of aspects provide useful information to anticipate the changes that are made by composing an aspect with a base specification. Moreover, I precisely define the way to recognize the category of an aspect according to its syntax. It can help the reasoning about the aspect by checking whether or not the intended category of an aspect consistent with the actual category of the aspect. Related results have been published in [ZKJ12a, ZKJ13, ZKJ11].

(3) The verification of aspectual composition in AO-PFA (Chapter 5): I present a technique to verify aspectual composition in AO-PFA. I define a set of validity criteria for PFA base specifications. I then present a set of definitions and propositions enabling the verification of the validity of aspects with regard to

their base specifications. Based on the proposed verification technique, we are able to verify the correctness of aspectual composition prior to the weaving process for aspects and base specifications. Results related to this topic have been published in [ZKJ12b, ZKJ11].

(4) The formalism of the weaving process in AO-PFA (Section 6.1): I specify the behaviour of a weaver used to generate the weaved specification w.r.t. a given aspect specification and a given PFA specification. I provide the formal representations for PFA specifications and aspect specifications, which are given as the input to the weaver. The mathematical structure of PFA specifications is expressed as a parametrized algebraic specification. The semantics of the pointcut corresponds to a quantification statement over the base specification. Related results have been published in [ZKJ13].

(5) Proofs for the convergence of the weaving process (Section 6.2): I construct a term rewriting system to solve the word problem that is engendered from the weaving process of AO-PFA. We prove the convergence of the weaving process by checking properties of the term rewriting system that I construct. A main result of term rewriting systems is that the word problem is decidable if the induced term writing system is convergent. I give the detailed proofs of the termination and confluence (i.e., convergence) of the rewriting system. I also prove the unambiguity of the weaving results by putting some restrictions on the form of join points that are selected by the aspects in AO-PFA. Results related to this topic have been published in [ZK13].

1.4 Related Publications

1.4.1 Journals

- [ZKJ13] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An Aspect-Oriented Language for Feature-Modeling. *Accepted May 2, 2013 for publication in Journal of Ambient Intelligence and Humanized Computing*, pages 15, 2013.

1.4.2 Referred Conferences and Workshops

- [ZKJ12a] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An aspect-oriented language for product family specification. In E. Shakshuki and M. Younas, editors, *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies*, volume 10 of Procedia Computer Science, ANT 2012 and MobiWIS 2012, pages 482 – 489, Niagara Falls, ON, Canada, August 2012.

- [ZKJ12b] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. Verification of aspectual composition in feature-modeling. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods, 10th International Conference, SEFM 2012*, volume 7504 of Lecture Notes of Computer Science, pages 109–125. Spinger Berline / Heidelberg, Thessaloniki, Greece, October 2012.

1.4.3 Technical Reports

- [ZKJ11] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An aspect-oriented language based on product family algebra: Aspects specification and verification. Technical Report CAS-11-08-RK, McMaster University, Hamilton, Ontario, Canada, November 2011. `http://www.cas.mcmaster.ca/cas/0template1.php?601`.

- [ZK13] Qinglei Zhang and Ridha Khedri. Proofs of the convergence of the rewriting system for the weaving of aspects in the AO-PFA language Technical Report CAS-13-01-RK, McMaster University, Hamilton, Ontario, Canada, April 2013. `http://www.cas.mcmaster.ca/cas/0template1.php?601`.

1.5 Structure of the Thesis

In this thesis, I proposed an approach that adapts the aspect-oriented paradigm to the feature-modeling level. Chapter 2 is a literature review of related work. Chapter 3 introduces the required mathematical and technical background of the thesis. The proposed aspect-oriented approach AO-PFA is discussed in Chapters 4, 5, and 6. In particular, Chapter 4 presents the syntax and usage of the proposed aspect-oriented specification language AO-PFA. Chapter 5 provides a formal technique for the verification of aspectual composition in AO-PFA. Chapter 6 discusses issues that are related to the automation of the weaving process for AO-PFA. Chapter 7 concludes our research and suggests the future work.

Chapter 2

Literature Review

In this chapter, we review several related works reported in the literature. Different techniques related to aspect-oriented software development are presented in Section 2.1. In Section 2.2, we explore known product family engineering techniques that use the aspect-oriented paradigm. Other techniques related to product family algebra are discussed in Section 2.3.

2.1 Aspect-Oriented Software Development

Software engineers continue seeking new paradigms to improve software quality. Although the object-oriented paradigm has been widely and successfully used during the past decades, researchers realize that there are limitations to decompose systems with one criterion as the object-oriented paradigm does. Around the middle of 90s [HL95], new design and programming techniques emerged with the aim of decomposing systems with more than one criteria. The most influential ones include SOP [HO93], CF [Ber94], AP [Lie96], and AspectJ [KLM+97]. We

have discussed AspectJ in Section 1.1.2 as an introduction to the aspect-oriented paradigm. Actually, all those new programming techniques mentioned above enable the modular management of crosscutting concerns and are later broadly referred to as aspect-oriented techniques [BA01, OT01, LOO01]. We discuss CF, SOP, and AP as mechanisms that support the aspect-oriented paradigm, and briefly compare them with the AspectJ language in Section 2.1.1. Although inspired by the programming level, the aspect-oriented paradigm was later adapted to the whole life-cycle of software development. We review different approaches for aspect-oriented requirement engineering, aspect-oriented architecture design, and aspect-oriented detailed design in Section 2.1.4, Section 2.1.3, and Section 2.1.2, respectively. We finally present several results regarding the assessment of the aspect-oriented paradigm in Section 2.1.6.

2.1.1 Aspect-Oriented Programming (AOP)

Subjective-Oriented Programming (SOP)

Subjective-Oriented Programming [HO93] is a programming paradigm where objects are described by composition of different subjective perspectives called *subjects*. SOP was later developed into MDSOC (Multi-Dimensional Separation of Concerns) [TOH+99], which focuses on the composition of different concerns. Hyper/J is an implementation of MDSOC for Java. In Hyper/J, each subject (or a dimension of concerns) is encapsulated by a module called a *hyperslice*. A concern is constructed from different subjects, and the composing rules for subjects are specified in *hypermodules*. The MDSOC mechanism intrinsically allows the encapsulation and composition of any type of concerns, including crosscutting concerns.

Furthermore, new concerns can be non-invasively introduced into the existing systems at any time by using MDSOC [OT01].

SOP and MDSOC also have a close relationship with aspect-oriented programming. The aspects can be specified by hyperslices while the composition of aspects can be specified by hypermodules. A key difference between the AspectJ and MDOSOC techniques is that MDSOC is a symmetric approach for all kinds of concerns, while the AspecJ is an asymmetric approach for base concerns and crosscutting concerns. In addition, unlike the forced composition of aspects in aspect-oriented languages, the composition of concerns in MDSOC is more flexible; MDSOC places the composition rules and concerns' behaviours in different modules.

Composition Filter (CF)

The CF [Ber94] model provides conventional object-oriented languages with a model extension, called CF classes. A CF class is an aggregation of several internal classes composed by filters. A *filter* is an instance of filter class and manages messages passing through it. We can specify conditions for accepting and rejecting the messages. If necessary, additional actions can be taken before actually executing methods of internal objects. The modularized filters can be used to encapsulate crosscutting concerns. Furthermore, as the filter is an enhancement attached to the internal objects, it can be added without changing the original objects. Crosscutting concerns such as inheritance, multiple views, synchronization and real time constraints, have been expressed with CF models [BA01].

In [HMM05], a concept mapping is constructed between the Composition Filter

model and aspect-oriented languages. Roughly speaking, the concept of pointcut corresponds to the declared elements in the filter and the concept of advice corresponds to the semantics of the filter. The simple declarative style makes the specification of filters independent from the underlying languages, which gives the CF models more flexibility for reusing a crosscutting concern. However, as the CF model only uses passing messages to interact between aspect and base concerns, it has less capability for composing a crosscutting concern.

Adaptive Programming (AP)

Adaptive Programming [Lie96] is a programming style that expresses the behaviours of a method by a *traversal strategy* and an *adaptive visitor*. Based on a class graph, the traversal strategy describes how to reach all participants of the method computations. With the traversal strategy, the adaptive method can capture all related participants of a crosscutting concern. The adaptive visitor defines the actions taken at each participant. With the adaptive visitor, the implementation of the crosscutting concern can be encapsulated in one place. Therefore, adaptive programming can be used to solve the tangling and scattering problems of crosscutting concerns. Besides, due to the use of reflection in many adaptive methods, the underlying language needs not be prepared for the additional actions specified by the adaptive visitor. Certain crosscutting concerns, such as synchronization and remote invocation have been implemented by adaptive methods.

Connecting with the aspect-oriented paradigm, the traversal strategy works as the point cut, whereas the adaptive visitor works as the advice. The DJ (Demerter/Java) library [HLM+98] is an adaptive programming Java package that

supports the aspect-oriented paradigm. As DJ library is a pure Java package, it is easier for an original Java programmer to implement the aspect-oriented paradigm. However, as the specifying of traversal strategy is based only on graphs, the types of crosscutting concerns that can be implemented by adaptive programming are limited.

Other Aspect-Oriented Programming Approaches

Besides developing new techniques for aspect-oriented programming, other types of aspect-oriented programming approaches focus on applying the aspect-oriented paradigm to the development of a specific domain. For example, the security concern is commonly recognized as an example of a crosscutting concern. The pointcut and advice constructors used in the aspect-oriented paradigm could implement security concerns in a more flexible and transparent way. When implementing a secure system, we should be aware that application developers are not expected to be security experts. Therefore, security concerns should be provided as a default. Moreover, the implementation of security concerns should allow the system managers to specify different security policies with different applications.

Shah et al. [SH04] implement an aspect oriented security framework, which categorizes security problems into two types. One type of security problems, such as buffer overruns and format vulnerabilities, is purely generated from implementation code, which can be handled directly with the aspect-oriented programming techniques, by which concrete security mechanisms and their composition with the target applications are implemented. The other type of security problems, such as protection of communication channels and event ordering enforcement, is

more complicated and more likely to be handled at the architecture and design levels. We also discuss later how to handle this type of security problems with early aspect-oriented approaches.

2.1.2 Aspect-Oriented Detailed Design (AODD)

The standard software detailed design approaches aim to design the necessary behaviours and structures for implementing the demanded systems. The crosscutting behaviours and structures at the detailed design level have the closest meaning as aspects in the implementation level. Adapting aspect-oriented programming to the detailed design level is almost straightforward, although there are some subtle differences. Those new design approaches to specify and compose crosscutting concerns at the design level are called aspect-oriented detailed design approaches, which support the modularization and composition of crosscutting concerns.

Although originally developed to support object-orientation design, UML is used as a basis by most of the aspect-oriented detailed design approaches. Most aspect-oriented detailed design approaches proposed in the literature are extensions of the conventional UML approaches. For example, SUP (State charts and UML Profile) [AEB03] uses UML profile to model base and aspect structure in class diagram. UFA (UML for Aspects) [Her02] specifies abstract UML packages, which are called aspect packages. An aspect package contains abstract classes and methods to support the symmetric aspect-oriented paradigm. Moreover, those aspect-oriented detailed design approaches are inspired by the mechanisms used in aspect-oriented programming, such as AspectJ or HyperJ. For example, AODM (Aspect Oriented Design Modeling) [SHU02] specifies an aspect using the stereotype “aspect”, which

implements the modularization and composition of aspects in an AspectJ-style. Theme/UML [BC04] is another design approach that specifies an aspect with a UML meta-model, which implements the modularization and composition of aspects in a HyperJ-style.

The above aspect-oriented detailed design approaches mainly focuses on the specifying and composition of aspects at the design level. However, they are developed for different levels of design. The high level approaches (e.g., [Her02]) are abstraction design without detail. Middle level approaches (e.g., [BC04]) are implementation-independent design with more details. Low level approaches (e.g., [SHU02]) are implementation platform specific. At each level of design, the main design tasks contain defining components, defining aspects, and defining compositions. In general, the aspect-oriented detailed design process is proposed as a refinement process to integrate design approaches of different abstraction level [batmaccRS$^+$05], starting from architectual design consequently to high level design, middle level design, and low level design. Compared with non-AO design, aspect-oriented detailed design can enhance the cohesiveness and reduce the coupling of modules for a system. Besides, aspect-oriented detailed design approaches provide means to resolve conflicts between different concerns at the design level. Furthermore, as a bridge between the architecture design and implementation stages, aspect-oriented detailed design approaches provide support for both backward and forward mapping of definite stages level crosscutting concerns to artifacts at other software development stages.

2.1.3 Aspect-Oriented Architecture Design (AOAD)

There are architectural aspects which inherently cannot be encapsulated by architectural abstractions and crosscut several components. However, the conventional architecture techniques do not explicitly recognize those architectural aspects. This shortage does not only lead to the ignorance of some potential aspects at the design and programming stages, but also leads to difficulties in deriving optimal architectures with satisfying qualities. To help handle those architectural aspects, new architect design approaches have been proposed, which are referred to as aspect-oriented architecture design.

Conventional architectual design approaches are adjusted to support aspect-oriented architectures. DAOP-ADL [PFT03] extends the ADL (Architectural Description Languages) with new terms "aspect" and "aspectEvaluationRule". The PCS (Perspective Concern Space) [Kan03] is a new architectural technique that supports the aspect-oriented paradigm by a conventional architectural approach, SADL (Structural Architecture Description Languages). Since SADL supports structural decomposition at multiple levels, it allows PCS to describe aspect-oriented architectures in a Hyper/J style. Another sort of approach are novel architectural approaches that are directly developed to support the aspect-oriented paradigm. For example, TranSAT [BCD+04] integrates a new concern by defining an architecture plane (the new concern), a join point mask and a set of transformation rules. This is an architectural level application of the join point designation model as used by AspectJ. AOGA (Aspect-Oriented Generative Approaches) [KGdLvS04] provides support for the identification and specification of architectural aspects. ASAAM (Aspect Software Architecture Analysis

Method) [Tek04] aims to identify the architectural aspects, evaluate the architecture with the crosscutting requirements, and provide information for architecture refactoring if necessary.

According to the above survey, one focus of aspect-oriented architecture design techniques is to identify and modularize aspects at the architectural level (e.g., [Tek04, KGdLvS04]). The architectual aspects together with the output artefact of aspect-oriented requirement engineering are taken as the input of the architecture design process. Aspect-oriented architectural description languages (e.g., [PFT03, Kan03]) then can be used to describe architecture in the aspect-oriented paradigm. Moreover, architectures can be evolved by aspect-oriented architectural evolution methods (e.g., [BCD+04, Tek04]). Same as at the requirement level, architectural aspects help to avoid crosscutting concerns remaining unhandled till the late design and implementation stages. Meanwhile, aspectual aspects enable the mapping of some crosscutting requirements directly to architectural constructors. Evaluation techniques for aspect-oriented architecture design complement the validation methods for crosscutting concerns between requirements and architecture design level.

2.1.4 Aspect-Oriented Requirement Engineering (AORE)

Since crosscutting requirements are recognised at the requirement level, we found conventional requirement engineering approaches fall short in dealing with them due to their tangling and scattering properties. To support the management of

crosscutting requirements, it is intuitive to adapt the existing concepts and mechanisms from the aspect-oriented programming techniques to the requirement level, which leads to a set of new requirement approaches that are referred to as aspect-oriented requirement engineering.

Many of the new requirement approaches are extensions of the conventional requirement techniques with explicit constructors for aspects. For example, AORE with Acrade [RMA03] is the first aspect-oriented requirement engineering approach that extends conventional viewpoint-based requirement engineering approaches with notions for aspect modularization and composition. ARGM (Aspects in Requirement Goal Model) [YLM04] uses goal-based requirement engineering approaches to identify and handle crosscutting requirements. Use-case based techniques, such as AOSD/UC (Aspect-Oriented Software Development with Use Cases) [JN04], and aspectual use case approach [AM03], modularize crosscutting functional requirements through *extended* and *included* use cases. Scenario modeling with aspects [WA04] is a scenario-based technique that models aspectual scenarios by IPS (Interaction Pattern Specifications) [kKFGS02]. Moreover, new requirement approaches are proposed in the light of other programming mechanisms that support the aspect-oriented paradigm. For examples, Cosmos and CORE (Concern-Oriented Requirement Engineering) [MAR05, MRA05] are inspired by the MDSOC mechanism introduced in the previous section.

The above brief survey of aspect-oriented requirement engineering approaches shows that the main issues in aspect-oriented requirement engineering are about the identification of crosscutting requirements, the treatment of both functional and non-functional requirements, the support of requirements composition (include

trade-off resolution), and the support of tracing crosscutting concerns. An initial process of aspect-oriented requirement engineering can be described as a sequence of *concern elicitation*, *concern identification*, *concern representation*, *composition and trade-off resolution*, and *requirement mapping* [batmaccRS+05]. In particular, we highlight three main benefits of aspect-oriented requirement engineering. First, the requirement-level composition of crosscutting concerns helps to early identify potential conflicts and trade-off resolution. Second, aspects modularization facilitates the tracing and mapping of crosscutting concerns to the later development stages. Last, the separation of crosscutting concerns eases the verification and validation of the whole system.

2.1.5 Verification Techniques for Aspectual Composition

An important topic related to my work is verifying the correctness of aspectual composition in aspect-oriented software development. For any reasonable size of systems, no existing tools and methods are capable to prove the compatibility of aspects and base systems for all possible weaving scenarios [Kat05]. The problem is usually confined to the verification of specific properties and/or specific weaving cases. In general, two groups of properties need to be verified for aspectual composition: inheritance properties and imposition properties [Sip03]. Inheritance properties refer to desired properties of the base system that should be maintained after the aspectual composition, while imposition properties refer to new properties that should be added by composing the aspect. In this section, we discuss several verification techniques that are commonly used in the literature for the verification of aspectual composition.

Static code analysis

Static analysis approaches such as typing and dataflow techniques are used to analyze the changes that can be made by an aspect on the base systems. The interferences of aspects are checked by analyzing the interaction between aspects when they are applied at the same join points. Generally speaking, it is possible to only analyze an aspect for all possible weaving scenarios when the aspect clearly declares parameters and variables used for waving. On the other hand, it is necessary to analyze both the aspect and the base system for each weaving when arbitrary bidding between the aspect and the base system is allowed (e.g., using same name in both code segments).

A collection of static analysis work focuses on the classification of aspects based on their effects on base systems [KG99, RSB04]. Identifying categories of aspects enable a way to prevent the harm of aspects, since certain categories of aspects will automatically ensure the preservation of some classes of properties of the base systems. The first relevant work by Katz and Cil [KG99] suggests three basic categories of aspects: spectative, regulative and invasive aspects. Later on, [RSB04] proposes a refined classification system for aspects according to an extensive static code analysis. The work related to what properties can be preserved by different categories of aspects is still at relatively early stages. An intuitive theorem of such classification is that all safety and liveness properties of the base system without next-state temporal modality are preserved by spectative aspects [SK03]. Moreover, the work of Katz in [Kat06] considers the problem regarding temporal logic properties.

Deductive proofs with assume–guarantee style

The correctness of aspecutal composition can also be verified according to deductive proofs over aspects. In particular, the assume–guarantee style is used in such approaches. The assumption specifies the properties to hold before the execution of the advice, while the guarantee asserts properties should hold after the execution of the advice. Existing tools, such as PVS [OSRSC01], SPIN [IC08] are used for the automatic proving process. Ideally, it is possible to have a deductive proof over an aspect once for all possible weaving. The assume-guarantee style may be used to verify both inheritance properties and imposition properties for aspectual composition. For the first group of properties, the guarantee assertion specifies properties to be added by the aspect. A compatible base system w.r.t. the given aspect should satisfy the associated assumptions at the join points. With regard to the inheritance properties, the key is to specify the invariant $\{I\}A\{I\}$ where I indicates inheritance properties that should be maintained before and after applying the aspect A. In some cases, if the invariant does not deductively hold after each step of the aspect, new restrictions need to be added to the invariant and new proofs need to be done for different base systems.

The idea of using deductive proofs for aspects was first described in [Dev03] and [Sip03]. In [Sip03], base systems are formalized as modular transition systems, and aspects are formalized as transformations that map a modular transition system into another modular transition system. In [Dev03], aspect-oriented programs are formalized based on ATS (Alternating Transition Systems), and both inheritance and imposition properties can be verified according compositional reasoning. In [Kat04, Kat06], Katz discussed how to ensure specific properties to be harmless

using deductive proofs. The work of [IC08] formalizes the behaviour of aspects by Promela, and focuses on the verification of imposition properties.

Model checking

Model checking is a technique widely used to automatically verify the correctness of finite-state systems. Formally, it is the problem of determining if a model of a system satisfies a property: searching if $M, s \models \phi$ hold, where M represents a state transition system, s is a state, and ϕ is typically written in temporal logic. There exist well used tools for model checking, such as SPIN and SMV, Bandera and SLAM. With respect to aspectual composition, the property that needs to be verified corresponds to ϕ, and base/weaved systems are formalized as M. The difficulty to check aspects and base systems separately is to formalize the effects of aspects on the original state machines of base systems.

The model checking approaches can be applied to the verification of aspects in various ways. An early work of [KFG04] proposed a method for modular verification of aspects related to a fix set of pointcut designators. An interface from the fix poincut designators is generated and fed to the fragment of the advice to verify the preservation of properties by model checking. A potentially attractive approach for verifying the correctness of aspectual composition independently of any base systems is mentioned in [SK03]. The solution is to write an abstraction of a base system, which is called a dummy basic program. Aspects are composed with the dummy basic program, and model checking is used to verify if the desired properties hold in the weaved system. Moreover, aspect-oriented-based model checking is another branch of related work [SK03, UT02]. The idea is to express

the crosscutting properties by aspects, and model checking approaches are then used to check the correctness of the weaved systems.

2.1.6 Assessment of Aspect-Oriented Approaches

As a new emerging software development methodology, no mature enough theoretical assessment methods exist for the evaluation of the aspect-oriented paradigm. To evaluate the effectiveness of the aspect-oriented approaches, assessments are taken by comparing aspect-oriented approaches with traditional approaches. The comparison methods include interview surveys, case-studies and experiments [PC01]. In [Bra03], the benefit of the aspect-oriented paradigm is assessed by interviewing system developers using aspect-oriented techniques. It shows that although current aspect-oriented approaches have limitations on the understandability, readability, testability and correctness of software in real world projects, the major factor of those limitations is the immaturity of those approaches. Therefore, aspect-oriented approaches still have the potential to improve the quality of software.

Empirical case-studies have been taken to compare aspect-oriented approaches with other methods, particularly, object-oriented approaches. Compared with the interview survey method, the case study assessment methods can give us more concrete evaluation results; however, it is often limited to the analysis of only the potentiality of aspect-oriented approaches in a particular domain of applications. For instance, Jaakko et al. [KT09] presents a case study of operating systems. It shows that when operating systems are developed using an aspect-oriented approach, the systems can be maintained easily. However the size of the code increases and its performance is degraded. Nevertheless, the size of the code and the performance of

the system are not as important in the domain of non-real time operating systems.

[PC01] describes a TCS (Thermic Control Simulation) system. The evaluation results indicate an increase in reusability and adaptability using aspect-oriented approaches when compared to standard object-oriented approaches. Similar to case-study methods, experimental assessment methods can give us numeric evaluation data. Moreover, it depends less on the domain of applications. However, the measure metric is often difficult to identify. The work presented in [KKG07] analyses the maintainability of software by making changes at code level. The experimental results suggest that, for a single change at code level, the aspect-oriented approach has less impact on other modules. In other words, they can improve maintainability, as changeability is a subclass attribute of maintainability.

2.2 Product Family Engineering Using the Aspect-Oriented Paradigm

The idea of using the aspect-oriented paradigm to the whole life-cycle of the product family engineering was proposed years ago [Gri00b]. However, earlier techniques that use the aspect-oriented paradigm in product family engineering mainly focus on the implementation level, such as Aspectual Mixin Layer [ALS06, AB06] and Caesar [MO03, MO04]. Aspectual Mixin Layers [AB06] explains how to combine aspect-oriented programming and feature-oriented programming approaches for implementing features. It allows one to use constructors from both feature-oriented programming (i.e., mixin class) and aspect-oriented programming (i.e.,

pointcut and advice). The mixin classes are still used to implement features, while aspects are used to capture the homogeneous and dynamic crosscutting characters of features. Caesar [MO03] is another approach that combines the supports of multi-abstraction modules and join point interception from feature-oriented programming and aspect-oriented programming languages respectively. In [MO04], Mezini et al. demonstrate that the Caesar language can be beneficial to the variability management in the context of product families. Only recently, some techniques are articulated to adapt the aspect-oriented paradigm at the earlier analysis and design stages. VML4RE [ASM+09] suggests a requirement specification language to compose elements from different requirement models. The concrete models supported by the language are use cases, interaction diagrams, and goal models. Xweave [GV07] is a model weaver supporting the composition of different views, which helps weaving variable parts of architectural models with base models. In VML4RE and Xweave, the variabilities of product families are composed with the concrete models of product families using the aspect-oriented paradigm.

In the literature, there are quite a few attempts which try to adapt the aspect-oriented paradigm at the feature-modeling level. While the domain engineering is not aspect free [RM06], adapting the aspect-oriented paradigm at the feature-modeling level fills the research gap in the literature. The above mentioned aspect-oriented techniques for product families can be seen as complementary techniques of the proposed work. By appropriately mapping mechanisms for aspects, we aim to handle aspects consistently and systematically from the feature models to the concrete models, and to the implementations of product families.

Besides the benefits and the necessity of adopting the aspect-oriented paradigm

at the feature-modeling level, we also conjecture that it is feasible to use the aspect-oriented paradigm at the feature-modeling level. As mentioned at the end of Section 2.1.6, the aspect-oriented paradigm can improve the modifiability of systems, but may also hinder the performance of systems. At the feature-modeling level, we can take the advantage of the aspect-oriented paradigm as at the programming level. Moreover, the performance issue is not as critical at the feature-modeling level as at the programming level. Therefore, the aspect-oriented paradigm is still reasonable in our context despite of its weakness at the programming level.

2.3 Feature-Modeling Related to Product Family Algebra

Product family algebra is the underling technique that we use in our research to specify feature models. There are several notations of feature models in the literature, such as FODA [KCH+90], FORM [KKL+01], FOPLE [KLD02], FeatuRSEB [GFd98], Generative Programming [Cza98], FORE [Str04], the Riebisch Technique [RBSP02], the van Gurp Technique [vGBS01], PLUSS [EBB05], etc.. In this thesis, we use a formal technique called *product family algebra* [HKM06, HKM08, HKM11a]. Comparing with other feature-modeling techniques, product family algebra is notable for the capability of formally and concisely specifying feature models. Product family algebra is able to capture a set of different notations found in current feature-modeling techniques. A thorough discussion on how other feature-modeling techniques can be easily translated into product family algebra is given in [AK10].

Moreover, a tool *Jory* [AK10] has been developed to implement automatic analysis of product family algebra-based specifications. One main issue in feature-modeling is to analyze the properties of a feature model (e.g., how many variants a family can generate), and the relations between different feature models (e.g., identical, subfamily and refinement). The analysis of feature models, especially for large-scale feature models, is an error-prone and tedious task. To support the automatic analysis of feature models, more accurate definitions and more efficient approaches are demanded in the feature-modeling community [BSRC10].

In the literature, three of the primary techniques used to support the automatic analysis of feature models are propositional logic [Bat05, MWCC08], constraint programming [BTRC05], and description logic [FZ06, WLS+07]. The basic idea for supporting the automatic analysis of feature models is firstly to translate feature models into certain formal or rigorous representations, and then to reason directly about those formal representations of feature models. With propositional logic based approaches, feature models are translated into propositional logic representations, and SAT solvers or other techniques are used to reason on feature models. Constraint programming based approaches can be considered as the extension of propositional logic approaches, which allow one to specify more flexible constraints on feature models. A feature model can be mapped into a CSP (Constraint Satisfaction Problem), and the analysis is conducted by a CSP solver. Description logic based approaches are also more expressive than propositional logic based approaches, and description logic reasoners such as RACER and Pellter are used to perform automatic analysis on feature models.

In our context, the feature models are translated into product family algebra-based specifications, and the specification is further analyzed by the tool *Jory* [AK10]. Comparing our product family algebra-based technique with the other techniques, the propositional logic actually can be considered as a particular model of the abstract structure of product family algebra. The requirement relation defined in product family algebra also can be used to specify constraints of feature models in a more flexible way. To implement the automatic analysis of feature models, *Jory* uses the binary decision diagrams provided by the library BuDDy [LN10], where one node in a binary decision diagram represents one feature. BuDDY can handle up to $50,000$ nodes in every megabyte of memory (tested for 2^{32} nodes) [AK10]. In other words, the *Jory* tool can provide us an effective and efficient way to automatically analyze feature models with a substantial number of features. Therefore, techniques based on product family algebra can take the advantages of representing feature models concisely and analyzing feature models automatically. More theoretical detail on product family algebra is given in Section 3.1.

2.4 Conclusion

In this chapter, a survey of the literature on aspect-oriented techniques at different stages of the software development was presented. The survey provides us with some insight on how to adapt the aspect-oriented paradigm to the feature-modeling level. Meanwhile, by exploring existing aspect-oriented techniques at different levels of product family engineering, we conjecture that it is necessary and feasible to adopt the aspect-oriented paradigm at the feature-modeling level. Finally, by comparing to other techniques for the modeling and analysis of feature models, we

argue the advantages of proposing techniques based on product family algebra.

Chapter 3

Background

In this chapter, we review necessary background for the thesis. Section 3.1 gives the details on product family algebra. Sections 3.2—3.5 provide mathematical knowledge that is needed for the research. Section 3.6 is a summary of how background knowledge is employed in the reminder of thesis.

3.1 Product Family Algebra

In this section, we first illustrate some basic notations of the feature models using a simple example of a product family. After that, we present the precise definition of product family algebra, and then define the basic notations of feature models based on product family algebra.

3.1.1 Basic Concepts of Feature Models

Figure 3.1 gives an example of a feature model adapted from a family of mobile phones. The feature model is represented using widely used feature diagram notations, and is basically a tree-like diagram. The example is complex enough to illustrate the basic characteristic concepts that are used by the feature-modeling community.

The root of the tree, which is a family of mobile phones, denotes the domain that is modeled. All other nodes express features that can be reused in the domain of mobile phones, and different types of edges constrain the combination of features for producing a valid mobile phone. For example, every phone must support `Calls` and `Screen_display`, which are referred to as *mandatory features*. But a phone does not necessarily include `GPS` and `Media_device`, which are referred to as *optional features*. Moreover, three alternative features, `Basic`, `Colour`, and `High_resolution` can be considered as specializations of the feature `Screen_display`. The term "*alternative features*" indicates that only one of the three features, `Basic`, `Colour`, and `High_resolution`, can be supported by a mobile phone. On the other hand, a mobile phone with the support of `Media_device` can include `Camera`, `MP3` player, or both of them. The features `Camera` and `MP3` are referred to as *or-group features*.

In addition, a feature model can be equipped with what are called cross-tree constraints. Those cross-tree constraints between features are typically in two forms: a feature A *requires* a feature B, or a feature A *excludes* a feature B. For instance, a mobile phone supports the `Camera` device must support the

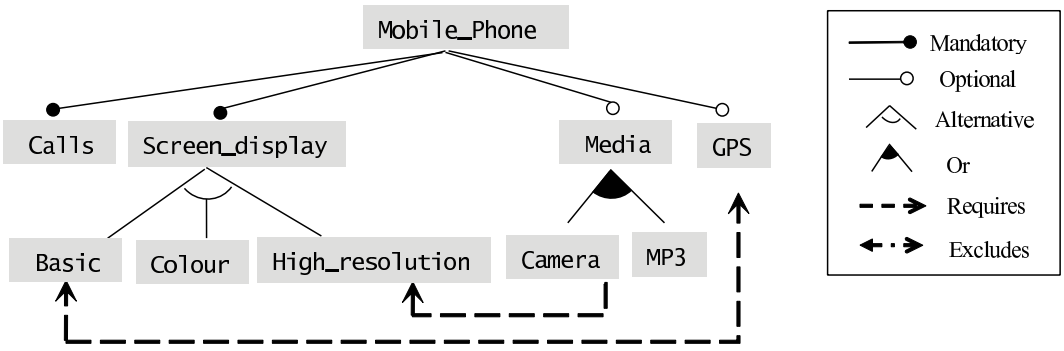

Figure 3.1: A feature model example adapted from [BSRC10]

`High_resolution` screen. The `GPS` and the `Basic` screen are incompatible features, which means both features `GPS` and `Basic` cannot be in a same product of mobile phone.

3.1.2 Mathematical Definitions of Product Family Algebra

We first introduce several properties on a binary operation, and give the definition of semirings.

Definition 3.1. *Given a binary operation $*$ on a set S, we say that $*$ is:*

- *Closed iff* $\forall(a, b \mid a, b \in S : a * b \in S)$;
- *Associative iff* $\forall(a, b, c \mid a, b, c \in S : (a * b) * c = a * (b * c))$;
- *Commutative iff* $\forall(a, b \mid a, b \in S : a * b = b * a)$;
- *Idempotent iff* $\forall(a \mid a \in S : a * a = a)$.

In particular, the operator $*$ has an identity iff $\exists(e \mid e \in S : \forall(a \mid a \in S : a * e = e * a = a))$, *and it has an annihilator iff* $\exists(e \mid e \in S : \forall(a \mid a \in S : a * e = e * a = e))$.

Definition 3.2. *Given two binary operation $*$ and $\circ$ on a set S, we say that $*$ is left-distributive (right-distributive) over $\circ$ iff* $\forall(a, b, c \mid a, b, c \in S : a * (b \circ c) = (a * b) \circ (a * c))$ *(iff* $\forall(a, b, c \mid a, b, c \in S : (a \circ b) * c = (a * c) \circ (b * c))$*).*

Definition 3.3 (semiring)**.** *A* semiring *is an algebraic structure denoted by a quintuple $(S, +, \cdot, 0, 1)$, such that S is a set, $+$ and $\cdot$ are binary operations over S, and $0, 1 \in S$. In particular, the binary operation $+$, called addition, is closed, associative, and commutative, and has an identity element 0. The binary operation $\cdot$, called multiplication, is closed and associative, and has an identity element 1. Multiplication is left and right distributive over addition. Moreover, 0 is the annihilator element for multiplication.*

Definition 3.4 (commutative and idempotent semiring)**.** *A commutative and idempotent semiring is a semiring $(S, +, \cdot, 0, 1)$ such that multiplication is commutative, and addition $(+)$ is idempotent.*

Definition 3.5 (product family algebra e.g., [HKM11a])**.** *A product family algebra is a commutative idempotent semiring $(S, +, \cdot, 0, 1)$, where*

(a) S corresponds to a set of product families;

(b) $+$ is interpreted as the alternative choice between two product families;

(c) $\cdot$ is interpreted as a mandatory composition of two product families;

(d) 0 corresponds to an empty product family;

(e) 1 corresponds to a product family consisting of only a pseudo-product which has no features.

Regarding the above definition of product family algebra, an equation e.g., $f_1 \cdot f_2 = 0$ indicates that the composition of the features f_1 and f_2 generates an empty family. Such equations can be used to reflect the fact that not all feature compositions are possible or desirable in reality. Moreover, an optional feature f can be interpreted as an alternative choice between the features f and 1. All those basic concepts as introduced in Section 3.1.1 can be expressed mathematically. For example, the product family of `Mobile Phone` can be represented by the term `Call` $\cdot$ `Screen_display` $\cdot$ (1+`Media`) $\cdot$ (1+`GPS`), which is referred to as a *product family term*. The term (`Basic` +`Colour`+`High_resolution`) represents alternative features for `Screen_display`, while the term (`Camera + MP3 + Camera` $\cdot$ `MP3`) represents the or-group features for `Media`. Both types of cross-tree constraints in feature models can be captured by a *requirement* relation over the product family algebra. To give the definition of the requirement relation, we also define two other relations over the product family algebra, *subfamily* and *refinement*.

Definition 3.6 (subfamily e.g., [HKM11a])**.** *For elements a and b in a product family algebra, the subfamily relation ($\leq$) is defined as* $a \leq b \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad a + b = b$.

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. Notice that a product family term can also be used to specify a configuration of a product family. We can check whether a configuration is a valid by using the subfamily relationship. Take the above mobile phone example and let `Camera`$\cdot$`MP3` correspond a certain configuration. Since we have (`Camera` $\cdot$ `MP3`) $\leq$ `Media`, we say `Camera` $\cdot$ `MP3` is a valid configuration of the feature model corresponding to `Media`.

Definition 3.7 (refinement e.g., [HKM11a])**.** *For elements a and b in a product*

family algebra, the refinement relation ($\sqsubseteq$) is defined as $a \sqsubseteq b \overset{\text{def}}{\Longleftrightarrow} \exists(c \mid: a \leq b \cdot c)$.

The refinement relation indicates that, for two given product families a and b, a is a refinement of b if and only if every product in family a has at least all the features of some products in family b. Taking the mobile phone example again, we have $\texttt{Media} \sqsubseteq (\texttt{Camera} + 1)$, which indicates that `Media` is a refinement of $\texttt{Camera} + 1$.

Definition 3.8 (requirement e.g., [HKM11a])**.** *For elements a, b, c, d and a product p in product family algebra, the requirement relation ($\rightarrow$) is defined in a family-induction style as:*

$$a \overset{p}{\rightarrow} b \quad \Longleftrightarrow_{df} \quad p \sqsubseteq a \Longrightarrow p \sqsubseteq b$$
$$a \overset{c+d}{\rightarrow} b \quad \Longleftrightarrow_{df} \quad a \overset{c}{\rightarrow} b \wedge a \overset{d}{\rightarrow} b$$

The requirement relation is used to specify constraints on product families. For elements a, b and c, $a \overset{c}{\rightarrow} b$ can be read as "a requires b within c". For example, the " requires" and "excludes" constraints in Figure 3.1 can respectively be represented as follows:

$$\texttt{Camera} \xrightarrow{\texttt{Mobile\_Phone}} \texttt{High\_resolution};$$
$$\texttt{Basic} \cdot \texttt{GPS} \xrightarrow{\texttt{Mobile\_Phone}} 0.$$

For more details on the use of this mathematical framework to specify product families, I refer the reader to [HKM06, HKM08, HKM11a].

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

Figure 3.2: Grammar of PFA language given in BNF notation

3.1.3 Tool Support

As we have mentioned, the tool *Jory* is developed based on product family algebra to represent and analyze feature models. *Jory* specifies feature models as PFA specifications, which are taken as the base specifications in the proposed aspect-oriented language. We give the grammar of PFA language in Figure 3.2. I use '\n' to denote the end of the line. Briefly speaking, there are three types of specification constructs in a PFA specification: basic feature declarations, labelled family equations, and constraints. Each basic feature is declared with a *basic feature label* preceded by the keyword *bf*. Each labeled family is defined as an equation with a *product family label* at the left side and a product family algebra term at the right side. A constraint is represented by a triple preceded by the keyword *constraint* and corresponds to a requirement relation in product family algebra.

Specification 1: A Mobile_Phone Product Family

```
    % Declarations of basic features
1.  bf Calls
2.  bf Basic
3.  bf Colour
4.  bf High_resolution
5.  bf Camera
6.  bf MP3
7.  bf GPS
    % Definitions of labeled product families
8.  Screen_display = Basic + Colour + High_resolution      % Specify alternative features
9.  Media = Camera + MP3 + Camera · MP3                    % Specify or-group features
10. Mobile_Phone = Calls · Screen_display · (1 + Media) · (1 + GPS)   % Specifying mandatory and optional features
    % Articulating a constraint on the family
11. constraint(Camera, Mobile_Phone, High_resolution)      % Specify a "requires" constraint
12. constraint(Basic · GPS, Mobile_Phone, 0)               % Specify a "excludes" constraint
```

Figure 3.3: Example of a PFA specification

Figure 3.3 is a PFA specification corresponding to the example of mobile phone product family. In this specification, Lines 1–7 declare basic features in the family, which corresponds to external notes of the diagram in Figure 3.1. Lines 8–10 define labeled product families, which corresponds to the internal notes in the diagram in Figure 3.1. Specially, the labeled product family in Line 10 is corresponding to the root of the diagram in Figure 3.1. Line 11–12 respectively specify the "requires" and "excludes" constraints given in Figure 3.1.

3.2 Needed Notions from Graph Theory

Graphs are mathematical structures that are wildly used to model pairwise relations between objects. In our research, we apply the graph theory [Die05] to detect invalid aspecutal composition. Meanwhile, the basic notations of graphs are mentioned in the diagram of feature models. We briefly introduce several required graph concepts in this subsection.

Definition 3.9 (graph)**.** *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.*

The degree of a vertex is the number of edges at the vertex. The vertices of degree 1 are called *external notes*, while other vertices are called *internal notes*.

A *digraph* is a graph where each edge has the *head* and the *tail* vertices. In particular, let (u, v) denote an edge in a digraph, then u is called the tail and v is called the head of the edge. When $u = v$, we say that the edge (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 $G = (V, E)$ is a non-empty list $v_0, e_0, v_1, ..., v_{k-1}, e_{k-1}, v_k$ where $v_i \in V$ for $i = 0, \ldots, k$, and $e_i \in E \wedge e_i = \{v_i, v_{i+1}\}$ for $i = 0, \ldots, k-1$. A *path* in a graph 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 denotes the length of the path. In particular, we say a (u, v)–path is a *cycle* when $u = v$.

3.3 Needed Notions from Universal Algebra

The concept of algebra is necessary for understanding both the underlying technique of a product family algebra and the proposed techniques in this thesis. Basic notations of universal algebra are given in this subsection.

Definition 3.10 (signature)**.** *A signature, denoted as $SIG = (S, OP)$, consists of a set of sorts S and a set of constants and operations $OP = \bigcup_{s \in S} K_s \cup \bigcup_{w \in S^+,\ s \in S} OP_{w,s}$.*

All sets K_s and $OP_{w,s}$ are pairwise disjoint sets such that K_s denotes the set of constant symbols of sort $s \in S$, and $OP_{w,s}$ denotes the set of operation symbols with argument sorts $w \in S^+$ and range sort $s \in S$.

Definition 3.11 (SIG-algebra)**.** *Let $SIG = (S, OP)$ be a signature. A SIG-algebra $\mathcal{A}$ consists of a carrier set $A = \bigcup_{s \in S} A_s$, and a mapping associated with each operation symbol $N : s_1, \ldots, s_n \to s$ in OP with a function $N_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$.*

Definition 3.12 (terms)**.** *Give a signature $SIG = (S, OP)$ and a set $X = \bigcup_{s \in S} X_s$ of variables disjoint from SIG. The sets of terms w.r.t. the sort s and X, denoted as $T_{OP,s}(X)$, is inductively defined as follows:*

- *Base cases: $X_s \cup K_s \subseteq T_{OP,s}(X)$*
- *Inductive cases: For all operation symbols $N \in OP$ such that $N : s_1, \ldots, s_n \to s$, we have $N(t_1, \ldots, t_n) \subseteq T_{OP,s}(X)$ where $t_1 \in T_{OP,s_1}(X), \ldots t_n \in T_{OP,s_n}(X)$.*

In particular, we denote $T_{SIG}(X) = \bigcup_{s \in S} T_{OP,s}(X)$, and denote the set of terms without variables (i.e., $T_{SIG}(\emptyset)$) as T_{SIG} for short.

Definition 3.13 (term algebra)**.** *Let $SIG = (S, OP)$ be a signature and X be a set of variables disjoint from SIG. The SIG-term algebra generated by X consists of the carrier set $T_{SIG}(X)$, and each operation symbol $N : s_1, \ldots, s_n \to s \in OP$ is interpreted as a function $N^{T_{SIG}(X)} : T_{OP,s_1}(X) \times \cdots \times T_{OP,s_n}(X) \to T_{OP,s}(X) : t_1, \ldots t_n \to N(t_1, \ldots t_n)$.*

Definition 3.14 (axiom)**.** *Give a signature SIG, and a set of variables X disjoint from SIG. An axiom w.r.t. SIG can be represented as a triple (X, L, R) where $L, R \in T_{SIG}(X)$. We call L the left-hand side (lhs) and R the right-hand side (rhs).*

3.4 Needed Notions from Algebraic Specifications

The algebraic specification [CEW93] is a formal description technique for abstract data types, which is developed to capture the specifications and designs of software systems. The theory of algebraic specifications is based on universal algebra and category theory. In this section, we introduce the basic concepts such as algebraic specifications, constraints, parametrized specifications. More theoretical foundations of algebraic specifications can be found in [EM85, EM90].

Definition 3.15 (algebraic specification [EM85]). *An algebraic (equational) specification $SPEC = (S, OP, E)$ consists of a signature $SIG = (S, OP)$ and a set of axioms E w.r.t. SIG.*

Definition 3.16 (parametrised specification [EM85]). *A parametrised specification is denoted as $PSPEC = (PAR, BOD)$, where PAR is called the formal parameter specification and BOD is called the body specification. In particular, let (S, OP, E) represent PAR and $(S1, OP1, E1)$ represent BOD, then an algebraic specification $(S, OP, E) \uplus (S1, OP1, E1)$ is called the target specification, where $\uplus$ stands for the disjoint union of sets.*

For a parametrized specification, the parameter specification is an algebraic specification, and it is a sub-specification of the target specification.

Definition 3.17 (algebraic specifications with constraints [EM90]). *An algebraic specification $SPEC = (S, OP, E)$ with a set of constraints C can be written as either $SPECC = (S, OP, E \cup C)$ or $SPECC = (S, OP, E, C)$, where C is in general a set of some first or higher order logical formulas that have to be satisfied by the algebra of $SPEC$.*

3.5 Needed Notions from Term Rewriting

Term rewriting [BN98] is a theoretical technique based on equational logic. Term rewriting can be applied in any context where efficient methods for equational reasoning are required. In this subsection, we briefly discuss the theory of term rewriting and present several main results. All of the following definitions are given in [BN98]. For lemmas, theorems, and corollaries listed here, we point out exactly where their proofs can be found.

3.5.1 Equational Problems

Definition 3.18 (semantic consequence)**.** *Given a set E of axioms w.r.t. a signature SIG, the SIG-algebra $\mathcal{A}$ is a model of E iff every axiom in E holds in $\mathcal{A}$. Let V be a countable infinite variable set disjoint from SIG, and $s, t \in T_{SIG}(V)$. We say an equation $s = t$ is a semantic consequence of E (i.e., $E \models (s = t)$) iff it holds in all models of E.*

Lemma 3.1 ([BP06, p. 8])**.** *Suppose Δ is a set of formulas and ψ and ρ are formulas, then* $(\Delta \cup \{\psi\}) \models \rho \iff \Delta \models (\psi \implies \rho)$

In particular, the relation $=_E \stackrel{\text{def}}{=} \{(s,t) \in T_{SIG}(V) \times T_{SIG}(V) \mid E \models s = t\}$ is called the *equational theory* induced by E. The relation $=_E$ is proved to be a congruence relation on $T_{SIG}(V)$, and $T_{SIG}(V)/_{=_E}$ denotes the quotient algebra w.r.t. $=_E$.

Definition 3.19 (word problem)**.** *Given a set E of axioms w.r.t. a signature SIG, and a set V of countable infinite variables disjoint from SIG, the word problem for E is the problem of deciding $s =_E t$ for arbitrary $s, t \in T_{SIG}(V)$.*

3.5.2 Reduction Relations

A reduction, usually denoted by $\longrightarrow$, is in general a binary relation on a set. We denote following notations for the reduction relation. Let $\overset{*}{\longrightarrow}$ denote the reflexive and transitive closure over $\longrightarrow$, and let $\overset{*}{\longleftrightarrow}$ denote the reflexive, transitive, and symmetric closure over $\longrightarrow$. We say x is *reducible* iff there is a $y \neq x$ such that $x \longrightarrow y$. We say y is a *normal form* of x iff $x \overset{*}{\longrightarrow} y$ and y is not reducible. Moreover, the normal form of x is denoted as $x\downarrow$ if x has a unique normal form, and x and y are *joinable*, denoted as $x \downarrow y$, if there is a z such that $x \overset{*}{\longrightarrow} z$ and $y \overset{*}{\longrightarrow} z$. In addition to the above notations, several important properties of the reduction relation that are used in our context are given in the following definition.

Definition 3.20. *A reduction relation $\longrightarrow$ is said to be confluent iff $x \overset{*}{\longrightarrow} y_1 \wedge x \overset{*}{\longrightarrow} y_2 \implies y_1 \downarrow y_2$. It is terminating iff every descending chain $a_0 \longrightarrow a_1 \longrightarrow \ldots$ is finite. It is convergent iff it is both confluent and terminating.*

Given a signature SIG and a countably infinite set V of variables disjoint from SIG, let $\mathcal{S}ub(T_{SIG}(V))$ or simply $\mathcal{S}ub$ denote a set of $T_{SIG}(V)$-substitutions which are a function $\theta : V \to T_{SIG}(V)$ such that $\theta(x) \neq x$ for only finitely many xs. Moreover, given a term $s \in T_{SIG}(V)$, the set of positions of the term s, denoted by $\mathcal{P}os(s)$, is inductively defined as follows:

- Base cases: If $s = x \in V$, then $\mathcal{P}os(s) = \{\epsilon\}$, where ϵ denotes the empty string.
- Inductive cases: If $s = f(s_1, \ldots, s_n)$, then

$$\mathcal{P}os(s) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(s_i)\}$$

For example, the set of position for term $\cdot(x, +(y, z))$ is $\{\epsilon\} \cup \{1 + \mathcal{P}os(x), 2 + \mathcal{P}os(+(y + z))\} = \{\epsilon, 1, 2 + \{\epsilon, 1 + \mathcal{P}os(y), 2 + \mathcal{P}os(z)\}\} = \{\epsilon, 1, 2, 21, 22\}$. On the other hand, given $p \in \mathcal{P}os(s)$, let $s|_p$ denote the subterm of s at position p. In particular, we define $s|_\epsilon = s$, and $f(s_1, ..., s_n)|_{iq} = s_i|_q$. For example, if we consider the position 22 and the term $\cdot(x, +(y, z))$, then we can obtain $\cdot(x, +(y, z))|_{22} = +(y, z)|_2 = z$.

Definition 3.21 (reduction relation induced by equations)**.** *Let E be a set of axioms w.r.t. the signature SIG and V be a countable infinite set of variables disjoint from SIG. The reduction relation induced by E, denoted by $\rightarrow_E \subseteq T_\Sigma(V) \times T_\Sigma(V)$, is defined as follows: $s \rightarrow_E t$ iff $\exists((l, r) \in E, p \in \mathcal{P}os(s), \theta \in \mathcal{S}ub \ | : \ (s|_p = \theta(l)) \wedge (t = s[\theta(r)]_p))$.*

Definition 3.22 (syntactic consequence)**.** *Give a set E of axioms w.r.t. the signature SIG. Let V be a set of countable infinite variables disjoint from SIG, and $s, t \in T_{SIG}(V)$. We say $s = t$ is a syntactic consequence of E ($E \vdash (s = t)$) iff we can derive $s = t$ by applying the inference rules which express the closure of E under reflexivity, symmetry, transitivity, substitutions, and SIG-operations.*

According to the definition of $\longrightarrow_E$, it is easy to show that $s \overset{*}{\longleftrightarrow}_E t$ iff $E \vdash (s = t)$. The following theorems are results that relate the semantic consequence to the syntactic one.

Theorem 3.2 ([BN98, p. 55])**.** *Let E be a set of axioms. The syntactic consequence relation $\overset{*}{\longleftrightarrow}_E$ coincides with the semantic consequence relation $=_E$.*

Theorem 3.3 ([BN98, p. 59])**.** *If E is finite and $\longrightarrow_E$ is convergent, then $=_E$ is decidable.*

3.5.3 Term Rewriting Systems

Definition 3.23 (rewrite rules)**.** *A rewrite rule w.r.t. a signature SIG is an axiom* (X, L, R) *w.r.t. a signature SIG such that* L *is not a variable and* $\mathcal{V}ar(L) \supseteq \mathcal{V}ar(R)$*. In this case we may present a rewrite rule as* $(X, L \longrightarrow R)$*.*

A redex (reducible expression) is an instance of the lhs of a rewrite rule, while rewriting the redex means replacing the redex with the corresponding instance of the rhs of the rule. Formally, given terms with variable declaration (t, Y_1) and (t', Y_2) and a rewrite rule $(X, L \longrightarrow R)$ w.r.t. a signature SIG, we say (t', Y_2) is derived from (t, Y_1) by application of $(X, L \longrightarrow R)$, if the two following conditions are satisfied:

1. There is $y_0 \in V$ and $t_0 \in T_{SIG}(Y_1 \cup Y_2 \cup \{y_0\})$ such that there is at most one occurrence of y_0 in t_0, and there is a mapping $\sigma : X \to T_{SIG}(Y_1 \cup Y_2)$, such that $t = \overline{h_1}(t_0)$, and $t' = \overline{h_2}(t_0)$ for

$$h_1(x) = \begin{cases} \overline{\sigma}(L) & \text{if } x = y_0 \\ x & \text{otherwise} \end{cases}$$

$$h_2(x) = \begin{cases} \overline{\sigma}(R) & \text{if } x = y_0 \\ x & \text{otherwise} \end{cases}$$

   where, $\overline{\sigma}$, $\overline{h_1}$, and $\overline{h_2}$ denote the extended substitutions of σ, h_1, and h_2, respectively.

2. $(X \cup Y_1) \neq \emptyset \implies T_{SIG}(Y_2) \neq \emptyset$.

We sometimes denote the derivation of terms by the substitution of terms. The

above derivation t' can be represented as $t' = t[\overline{L}/\overline{R}]$, where $\overline{L} = \overline{\sigma}(L)$, $\overline{R} = \overline{\sigma}(R)$, and $\overline{L}$ is subterm of t.

Given a set of rewrite rules R, we say that (t_r, Y_r) is derivable from (t_1, Y_1) with rules R, denoted as $(t_1, Y_1) \underset{R}{\longrightarrow} (t_r, Y_r)$, if there is a rewrite sequence (t_1, Y_1), (t_2, Y_2), ..., (t_r, Y_r) for $r > 1$ such that for $i = 1, 2, \ldots, r-1$, (t_{i+1}, Y_{i+1}) is derived from (t_i, Y_i) by application of some rule in R. The following theorem is a basic result of term rewriting systems, which is used in this thesis.

Theorem 3.4 ([EM85, p. 127]). *Let R and Q be an arbitrary sets of rewrite rules then $(t_1, X_1) \underset{R}{\longrightarrow} (t_2, X_2)$, and $(t_2, X_2) \underset{Q}{\longrightarrow} (t_3, X_3)$ imply $(t_1, X_1) \underset{R \cup Q}{\longrightarrow} (t_3, X_3)$.*

Definition 3.24 (abstract reduction system). *An abstract reduction system is a pair $(A, \longrightarrow)$, where $\longrightarrow$ is a reduction relation on the set A.*

A *term rewriting system (TRS)* R is a set of rewrite rules w.r.t. a signature SIG, which corresponds to a reduction system $(T_{SIG}(V), \longrightarrow_R)$. The reduction relation $\longrightarrow_R$ is well-defined as given in Definition 3.21. We say R is terminating, confluent, and convergent iff $\longrightarrow_R$ is terminating, confluent, and convergent. In the remainder of this subsection, we present several results related to term rewriting systems, which are used later to prove the termination and confluence of the term rewriting system in our context.

Definition 3.25 (lexicographic path order). *Let $>$ be a strict order over signature Σ. The lexicographic path order $>_{lpo}$ induced by $>$ is defined as follows: $s >_{lpo} t$ iff*

(LOP1) *$t \in \mathcal{V}ar(s)$ and $s \neq t$.*

(LOP2) *$s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$ and*

(LOP2a) $\exists(i \mid 1 \leq i \leq m : s_i \geq t)$

(LOP2b) $f > g \wedge \forall(j \mid 1 \leq j \leq n : s >_{lpo} t_j)$

(LOP2c) $f = g \wedge \forall(j \mid 1 \leq j \leq n : s >_{lpo} t_j)$
$\wedge \exists(i \mid 1 \leq i \leq m : s_1 = t_1, \ldots s_{i-1} = t_{i-1} \wedge s_i >_{lop} t_i)$

Theorem 3.5 ([BN98, p. 119]). *For any strict order $>$ over signature Σ, the induced lexicographic path order $>_{lpo}$ is a simplification order on $T_\Sigma(V)$.*

Theorem 3.6 ([BN98, p. 103]). *Let Σ be a finite signature. Every simplification order $>$ on $T_\Sigma(V)$ is a reduction order.*

Theorem 3.7 ([BN98, p. 103]). *A term rewriting system R terminates iff there exists a reduction order $>$ that satisfies $l > r$ for all $l \to r \in R$.*

Given a set of equations E, and two terms s and t, the process of finding a substitution θ such that $\theta(s) = \theta(t)$ is known as *"syntactic unification"*. We call θ a unifier of s and t, or a solution of the equation $s \stackrel{?}{=} t$.

Definition 3.26 (Most General Unifier). *Let S be a set of equation $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n)$. A substitution θ is a most general unifier of S iff (1) θ is a solution of S, i.e., $\forall(i \mid 1 \leq i \leq n : \theta(s_i) = \theta(t_i))$; (2) θ is a least element of all the solution or unifier of S, i.e., $\forall(\theta' \mid \theta'$ is a solution of $S : \exists(\delta \mid : \theta' = \delta\theta))$*

Definition 3.27 ([BN98, p. 139]). *Assume that two rules $l_1 \to r_1$ and $l_2 \to r_2$ satisfy $\mathcal{V}ar(l_1, r_1) \cap \mathcal{V}ar(l_2, r_2) = \emptyset$. Let $p \in \mathcal{P}os(l_1)$ such that $l_1|_p$ is not a variable and let θ be a mgu (most general unification) such that $\theta(l_1|_p) = \theta(l_2)$. This determines a critical pair $\langle \theta(r_1), \theta(l_1[\theta(r_2)]_p) \rangle$. If two rules give rise to a critical pair, we say that they overlap. When considering the overlap of a rule with a renamed variant of itself, it is safe to ignore the case where $p = \epsilon$.*

Lemma 3.8 ([BN98, p. 76]). *Let $s_1, \ldots, s_m$, and $t_1, \ldots, t_n$ be terms over the signature Σ. An equation $f(s_1, \ldots, s_m) \stackrel{?}{=} g(t_1, \ldots t_n)$, where $f, g \in \Sigma, f \neq g$, has no solution.*

Theorem 3.9 ([BN98, p. 140]). *A terminating term rewriting system is confluent iff all its critical pairs are joinable.*

3.6 Conclusion

In this chapter, we explained the technique of product family algebra for specifying feature models. Moreover, we introduced several graph theoretical concepts, elements of universal algebra, algebraic specifications, and term rewriting systems. The above concepts and notions are needed for the discussions presented in the remaining chapters of this thesis and for making the thesis self-contained. In Chapter 4, an aspect-oriented specification language is proposed by analogizing the essential elements of the aspect-oriented paradigm to the context of product family algebra. In Chapter 5, graph concepts are used to detect invalid aspectual composition. The concepts from universal algebra play an important role in the formal techniques of algebraic specifications and term rewriting systems. In Chapter 6, algebraic specifications and term rewriting systems are further discussed and used in the automation of the weaving process.

Chapter 4

Specifying Aspects with AO-PFA

In this chapter, we discuss how to articulate aspects in feature models and particularly how to properly use the construct of the proposed language AO-PFA. Section 4.1 describes the general design of the specification language. Section 4.2 precisely presents the syntax and usage of the proposed specification language AO-PFA. Section 4.3 presents a classification of aspects in AO-PFA. Section 4.4 gives examples to illustrate the usage of the proposed language. The conclusion is the subject of Section 4.5.

4.1 Introduction

The aspect-oriented paradigm can improve the modularity and modifiability of systems by decomposing systems with additional criteria. With the aspect-oriented paradigm, base concerns and crosscutting concerns of a system are separately encapsulated. The composition of base concerns is the symmetric composition, while the composition of aspect and base concerns, which is also referred to as aspectual

composition, is the asymmetric composition. The designed product family algebra-based specification language should facilitate both the symmetric and asymmetric composition of feature models. At the feature-modeling level, base concerns refer to different views of a feature model. As discussed in Chapter 1, constraints are used in product family algebra to tackle the view reconciliation problems for the systematic composition of feature models. The symmetric composition of feature models can be related to the view integration of feature models. The asymmetric composition of feature models, on the other hand, can be considered as introducing additional information to the feature model of the base concern.

4.1.1 Rationale for AO-PFA Design

The proposed aspect-oriented language is an extension of the PFA language with the ability to specify aspects, and thus is called AO-PFA. The choice of extending the PFA language enables us to reuse existing tools for product family algebra. Figure 4.1 conveys the general idea of base specifications, aspects, and their weaving results. We identify two types of specifications in AO-PFA: PFA specifications and aspect specifications. A PFA specification (See Section 3.1) takes the role as the base specification, which specifies the integrated views of a product family. Additional information is introduced to the base specification by weaving the aspect specification to the base specification. The resulting specification is still a PFA specification, which can be automatically and efficiently analyzed using the tool *Jory*.

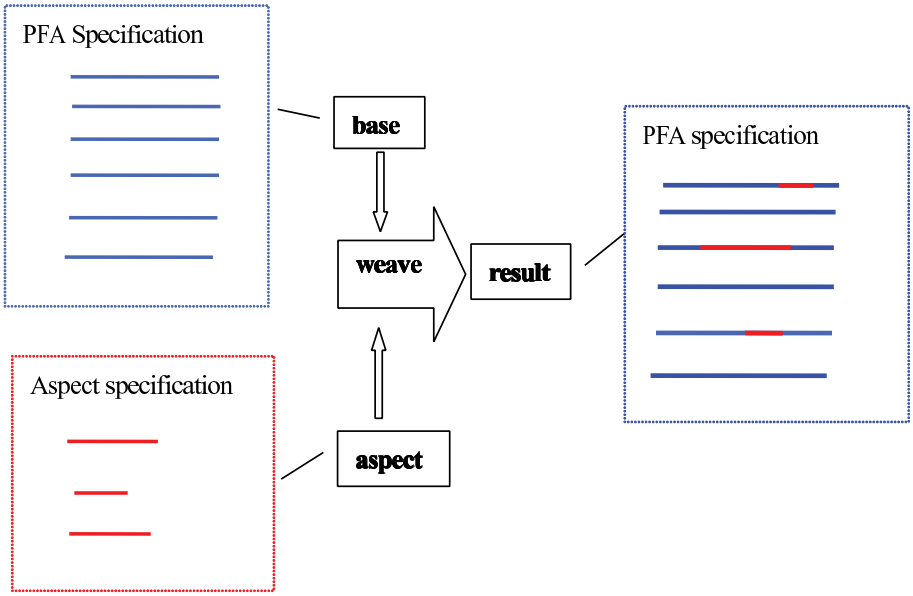

Figure 4.1: General Illustration of AO-PFA

To facilitate the asymmetric composition of feature models, several requirements should be considered regarding the design of aspect specification. We identify that an aspect generally requires to capture *what*, *where*, and *how* an aspect is composed with the base specification.

The *advice* in an aspect describes what is composed with the base specification. It requires that the type of base specifications is not changed by weaving aspects to the base specifications. Hence, the advice needs to be syntactically similar to the base specifications. In particular, the proposed language for advice should enable the introduction of product family algebra terms, which are basic elements of PFA specifications.

The *pointcut* in an aspect identifies locations where the advice is introduced in the base specification. We need to specify a quantification statement over the base specification, which is considered as a trigger for composing aspects. Specially,

corresponding to different types of constructs in base specifications, the proposed language for aspects should enable the matching of basic features, labeled families, and constraints in PFA specifications.

The composition rule for an aspect at the selected *join points* captures how the aspect is composed at the identified locations. Considering all possible changes that we want to impose on the base specification, the identified locations might be augmented, removed, or replaced. Therefore, the proposed language for aspects should also enable us to specify composition rules that handle the above three kinds of modification scenarios.

4.1.2 A Running Example

The following example of an elevator product family is adapted to illustrate the usage of the proposed language. The diagram in Figure 4.2 illustrates a feature model of `elevator_product_line`, which includes a mandatory feature `base_functionality` and an optional feature `service` that provides additional service for the elevator family. The `base_functionality` includes a mandatory feature `move_control` and an optional feature `light_display`. We consider two unanticipated variabilities for `service`: `light_reset` and `failure_capture`. The feature `light_reset` may inherently depend on the feature `light_display`, and the feature `failure_capture` may inherently depend on both the `move_control` and the `light_display`. In addition, we also assume that the feature `service` can be evolved with a mandatory feature `logging`.

Specification 1 in Figure 4.3 corresponds to a base (primary) specification for the above elevator product family. Lines 1–3 declare three basic features

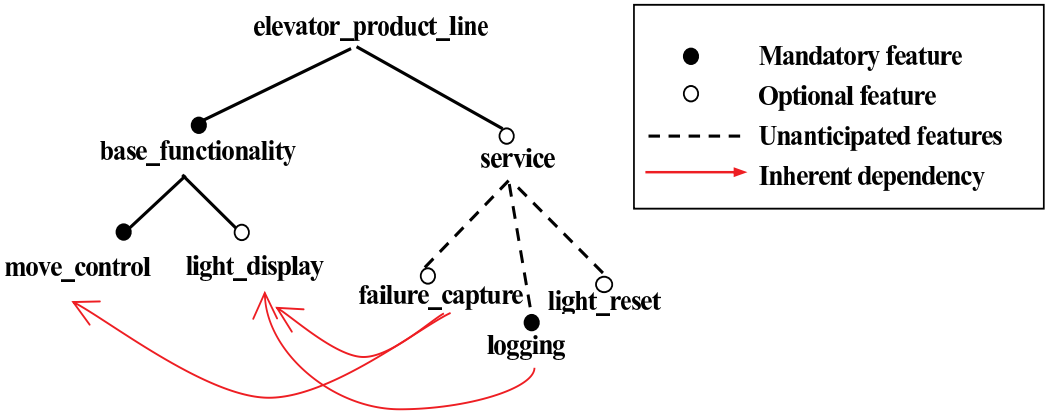

Figure 4.2: A simplified example of the feature model for an elevator system

Specification 1: A PFA specification of the elevator product family

1. bf move_control
2. bf light_display
3. bf service
4. optional_light_display = light_display +1
5. optional_service = service +1
6. base_functionality = move_control · optional_light_display
7. elevator_product_line = base_functionality · optional_service
8. full_base_functionality = move_control · light_display
9. customized_elevators = move_control +full_base_functionality · service
10. constraint(service, elevator_product_line, light_display)

Figure 4.3: Example of a base specification

named `base_functionality`, `light_display`, and `service`. The family equations in Line 4 and Line 5 indicate that `light_display` and `service` are optional features in the considered family. Line 6 and Line 7 give the definitions of `base_functionality` and `elevator_product_line`, respectively. Line 8 and Line 9 correspond to two specific configurations, named `full_base_functionality` and `customized_elevators`, of the feature models. Precisely, the configuration

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

Figure 4.4: The language for the specification of aspects

`full_base_functionality` is a member of `base_functionality`, and the configuration `customized_elevators` is a subfamily of `elevator_product_line`. Line 10 is a constraint, which indicates that within the product family `elevator_product_line` the feature `service` always requires `light_display`. The elevator example is used as a running example in the next section, where we explain how the PFA specification can be modified with additional features `light_reset`, `failure_capture`, and `logging` by composing aspects.

4.2 Aspect Specifications in AO-PFA

The grammar of an aspect specification in AO-PFA is given in Figure 4.4, where ϵ denotes the empty string. In general, there are a sequence of aspects in an aspect

specification, while each aspect is compactly specified as follows:

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

The syntax of an aspect contains three parts: a product family label $\langle \mathsf{aspectId} \rangle$, a product family term $\langle \mathsf{Advice(jp)} \rangle$, and a triple $(\langle \mathsf{scope} \rangle, \langle \mathsf{expression} \rangle, \langle \mathsf{kind} \rangle)$. The above syntax of an aspect allows one to specify the demanded language expressiveness that is identified in Section 4.1.1. The equation $\langle \mathsf{aspectId} \rangle = \langle \mathsf{Advice(jp)} \rangle$ corresponds to the *advice* of an aspect in AO-PFA. The equation sometimes is referred to as the advice equation in the reminder of this thesis. Moreover, a variable jp denotes an instance of *join points* that are selected by an aspect. Therefore, composition rules at the selected join points can be specified according to the appearance of jp in $\langle \mathsf{Advice(jp)} \rangle$. We further discuss different modification scenarios at the selected join points (i.e., augmenting, narrowing, and replacing) in Section 4.2.2. The triple $(\langle \mathsf{scope} \rangle, \langle \mathsf{expression} \rangle, \langle \mathsf{kind} \rangle)$ is the *pointcut* language in AO-PFA. Pointcuts associated with basic features, labeled families, and constraints are respectively presented in Section 4.2.3.

In the following of this section, the proposed language is further explained along with those essential elements for aspect-oriented techniques (i.e., join points, advice, and pointcut).

4.2.1 Join Points in AO-PFA

All product family terms defined in PFA specifications should be potential join points in the AO-PFA. More specifically, in a PFA specification there are two roles for the same form of a product family term; they are either being defined or being

referenced. For example, the family `base_functionality` in Figure 4.3 is being defined at the left-hand side in Line 6, while it is being referenced by another family at the right-hand side in Line 7. Consequently, we identify two types of join points, the definition join points and the reference join points. Syntactically speaking, the product family terms specified in feature declarations and at left-hand sides of family equations are definition join points, while product family terms specified in constraints and at the right-hand sides of family equations are reference joint-points.

It is necessary to distinguish the definition and reference join points at the feature-modeling level due to the semantic difference when integrating new aspects at these two types of join points. Introducing an advice at the identified definition join points affects the internal description of the join points, whereas introducing an advice at the identified reference join points does not. Actually, the latter case may affect internal descriptions of product families that include the join points. The definition join point can be considered as a white box whereas the reference joint point can be considered as a black box. When it comes to the detailed level of features, introducing advice at these two types of locations can cause very different results. These two types of join points are further referred to in the following discussion on the advice and the pointcut of aspects.

4.2.2 Advice in AO-PFA

In an aspect of AO-PFA, we use an equation $\langle \mathsf{aspectId} \rangle = \langle \mathsf{Advice(jp)} \rangle$ to specify the advice that is introduced at the selected join points. According to different types of pointcuts that are discussed in the next subsection, the selected join points

are either associated with the definition or reference of a specified product family algebra term. In particular, ⟨aspectId⟩ specifies new labels to rename the join points if the pointcut is associated with definition join points, while ⟨aspectId⟩ is always expressed as a variable jp if the pointcut is associated with reference join points. Moreover, based on the form of ⟨Advice(jp)⟩, we distinguish the following different modification scenarios at the selected join points:

Augmenting: The selected join points should still appear in the resulting specification. In the context of product family algebra, such an augmenting effect can be specified by a product family term constructed with a variable *jp* that represents an instance of the join points.

Narrowing: The selected join points are simply removed in the resulting specification. Such a narrowing effect corresponds to the constant element 1 of product family algebra, which represents a pseudo-product with no features.

Replacing: In this case, the selected join point is replaced by arbitrary product family terms that do not refer to the original join point. In other words, the advice should be in the form of a ground product family term (i.e., a term constructed without the variable jp).

4.2.3 Pointcuts in AO-PFA

By analogy to pointcut languages designed in other aspect-oriented techniques, the proposed pointcut language specifies three attributes to identify a set of join points in the PFA specification: the scope of join points, a predicate that characterizes the join points, and the exact matching pattern of join points. Recall that we express the pointcut language as a triple: (⟨scope⟩, ⟨expression⟩, ⟨kind⟩). Without

Table 4.1: Summary of types of pointcuts

Scope	
Default	*base*
Explicit Scope	through
	within
Excluded Scope	protect(⟨scope⟩)
Combined Scope	⟨scope_1 :scope_2⟩
	⟨scope_1;scope_2⟩

Expression	
Default	*true*
Explicit Expression	Boolean Expression

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

lose of clarity, we sometimes refer to the first component as the scope pointcut, the second component as the expression pointcut, and the third component as the kind pointcut. Table 4.1 summarizes the various types of each component of the pointcut triple.

Two types of join point scopes are designed: within and through. A scope of type within captures join points in specified lexical structures, while a scope of type through captures join points in a specified 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 in the union of the two specified scopes. Separating two scopes by ";" indicates that eligible join points are in the intersection of the two specified scopes. Moreover, we use protect 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 pointcut works as a guard for the selected join points. Precisely, we use Boolean expressions on the language of product family algebra to specify this component of the pointcut triple. The expression *true* is taken as default.

The kind pointcut is used to identify particular product family terms in the base specification. Unlike the scope pointcut and the expression pointcut, the kind

pointcut must be specified explicitly; there is no default value for the kind pointcut. As product family terms are associated with basic features, labeled families, and constraints, different types of kind pointcuts are further discussed in the reminder of this subsection (i.e., feature-related pointcut, family-related pointcut, and constraint-related pointcuts).

Feature-related pointcut

Two kinds of pointcuts, declaration and inclusion, are introduced to select join points associated with basic features in PFA specifications. The difference between these two kinds of pointcuts resides in whether or not the feature's definition can be changed. The declaration pointcut captures join points where a specific feature is declared. Figure 4.5 shows an example of the use the declaration pointcut. We specify an aspect in Figure 4.5(a) to express a requirement for introducing two optional features `failure_capture` and `light_reset` to the original definition of `service`. Taking Specification 1 (Figure 4.3) as our base specification, the pointcut here would capture a join point at Line 3 of Specification 1. Furthermore, as the scope pointcut is *base*, all references to the original `service` should be changed to the new one. Specification 2 in Figure 4.5(b) shows the result of weaving this aspect with Specification 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. The bold and italic fonts denote the modifications that are implied by changing the definition of basic features. Notice that the old feature `service` may be removed from the specification after weaving if there is no further reference to it within the whole specification.

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

(a) An aspect for modifying the definition of a basic feature

Specification 2: A PFA specification with new features `failure_capture` and `light_reset` that relate to the definition of `service`

1. bf move_control
2. bf light_display
3. bf *failure_capture*
4. bf *light_reset*
5. bf **service**
6. *service_new = (failure_capture + 1) · (light_reset + 1)*
7. optional_light_display = light_display + 1
8. optional_service = ***service_new*** + 1
9. base_functionality = move_control · optional_light_display
10. elevator_product_line = base_functionality · optional_service
11. full_base_functionality = move_control · light_display
12. customized_elevator = move_control+full_base_functionality · ***service_new***
13. constraint(***service_new***, elevator_product_line, light_display)

(b) A specification resulting from weaving the aspect in (a) with Specification 1

Figure 4.5: Example of using the declaration pointcut

The inclusion pointcut captures join points where a specific basic feature appears in any other product families. Figure 4.6 shows an example of using the inclusion pointcut. We specify an aspect in 4.6(a) to introduce a new feature `light_reset` in any product including `light_display`. Taking Specification 1 given in Figure 4.3 as our base specification again, the result of weaving this aspect is given by Specification 3 in Figure 4.6(b). As we can see, the new feature `light_reset` is introduced at the right-hand sides of both Line 4 and Line 8 in Specification 1, where the feature `light_display` is referenced.

As we have mentioned, the scope pointcut can be specified with a combination of the kind pointcut to express a further specific requirement. The example in

Aspect	jp = jp · light_reset
where	jp ∈ (*base*, *true*, inclusion(light_display))

(a) An aspect for modifying the reference of a basic feature

Specification 3: A PFA specification with a new feature `light_reset` that relates to the feature `light_display`

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

(b) A specification resulting from weaving the aspect in (a) with Specification 1

Figure 4.6: Example of using the inclusion pointcut

Figure 4.7 illustrates an aspect with the non-default scope pointcut and the non-default expression pointcut. Assume that inappropriate interactions are detected in the configuration `full_base_functionality`. The aspect in Figure 4.7(a) is for adding a new feature `failure_capture` in the family `customized_elevators` to capture all defective behaviours related to `move_control`. Considering Specification 1 as base specification, the kind pointcut inclusion(`move_control`) captures the reference of the feature `move_control`. The eligible join points are further bounded by the intersection of two scopes within(`customized_elevators`) and through(`full_base_functionality`). The within scope pointcut narrows the scope of join points to only Line 9 of Specification 1. Since the through scope pointcut specifies that the feature `move_control` should be from `full_base_functionality`, we do not compose `failure_capture` with the first `move_control` in Line 9 of Specification 1. Moreover, let $\#(a)$ represent the number of members in a family a. The expression pointcut $\#(jp) \geq 1$ indicates that there should be at least one

Aspect jp = jp · failure_capture
where jp ∈ (within(customized_elevators); through(full_base_functionality), #(*jp*) ≥ 1, inclusion(move_control))

(a) An aspect for bounding the scope and specifying a guard for join points

Specification 4: A PFA specification with a new feature `failure_capture` that relates to the `move_control`

bf *failure_capture*
...
9. customized_elevators = move_control + **move_control** · *failure_capture* · light_display · service
...

(b) A specification resulting from weaving the aspect in (a) with Specification 1

Figure 4.7: Example of using the non-default scope pointcut and the non-default expression pointcut

member in the corresponding family of the selected join points, which is satisfied by `full_base_functionality` in this case. The result of weaving this aspect is given in Specification 4 (Figure 4.7(b)).

Family-related pointcut

To capture join points associated with labeled families in PFA specifications, four kinds of pointcuts, creation, component_creation, component and equivalent_component, are designed. The creation and component_creation pointcuts are related to the definition join points, while the component and equivalent_component pointcuts are related to reference join points. Furthermore, the difference between the creation and component_creation pointcuts resides in whether we change the definition of a specified family directly or whether we change the definitions of its components. The difference between the component and equivalent_component pointcuts resides in whether the reference of a specified family is direct or indirect.

Aspect	jp_new = jp · logging
where	jp ∈ (protect(customized_elevators), true, creation(service_new))

(a) An aspect for modifying the definition of a labeled family

Specification 5: A PFA specification with new feature `logging` that relates to the definition of `service_new`

bf *logging*
...
5. **service_new** = (failure_capture + 1) · (light_reset + 1)
service_new_new = *service_new* · *logging*
...
8. optional_service = ***service_new_new*** + 1
...
12. customized_elevators = move_control + full_base_functionality · service_new
13. constraint(***service_new_new***, elevator_product_line, light_display)

(b) A specification resulting from weaving the aspect in (a) with Specification 2

Figure 4.8: Example of using the creation pointcut

The effects of the creation pointcut is quite similar to the declaration pointcut, excepting that the captured join points are labeled families instead of basic features. Taking Specification 2 in Figure 4.5 as our base specification, Figure 4.8 illustrates an example of using the creation pointcut. The aspect in Figure 4.8(a) is for introducing a new feature `logging` to the original definition of `service_new`. The resulting specification is illustrated by Specification 5 in Figure 4.8(b). Notice that `service_new` at Line 12 in Specification 5 is not changed to the new one since we specify a protect scope pointcut, which exemplifies a case that we want the legacy configuration `customized_elevators` to keep unchanged.

The component_creation pointcut also captures definition join points. However, unlike the creation pointcut, the component_creation pointcut refers to the definitions of all components in the specified families. Assume that we want to capture any defective behavior in the `full_base_functionality`. However,

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

(a) An aspect for modifying the definitions of components of a labeled family

Specification 6: A PFA specification with a new feature `failure_capture` that relates to component definitions of `full_base_functionality`

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*
   ...
4. optional_light_display = ***light_display_new*** + 1
   ...
6. base_functionality = ***move_control_new*** · optional_light_display
   ...
8. **full_base_functionality** = ***move_control_new*** · ***light_display_new***
   ...
10. constraint(service, elevator_product_line, ***light_display_new***)

(b) A specification resulting from weaving the aspect in (a) with Specification 1

Figure 4.9: Example of using the component_creation pointcut

`full_base_functionality` is composite and we cannot be sure which component might cause the defective behavior. Therefore, we add a feature `failure_capture` to each of its components, `move_control` and `light_display`. The aspect in Figure 4.9(a) specifies the above requirement, and the resulting specification of weaving the aspect with Specification 1 (Figure 4.3) is given in Specification 6 (Figure 4.9(b)). The captured join points are those definition join points at Line 1 and Line 2 of Specification 1. Correspondingly, all references to those components in Line 4, 6, 8, and 10 are changed to the new ones.

The component pointcut is similar to the inclusion pointcut. But the component pointcut refers to the appearance of the specified labeled families instead of basic features. Figure 4.10 illustrates an example of introducing the feature `logging` to

Aspect jp = jp · logging
where jp ∈ (base, true, component(service_new))

(a) An aspect for modifying the reference of a labeled family

Specification 7: A PFA specification with a new feature `logging` that relates to the reference of `service_new`

bf *logging*
...
8. optional_service = **service_new** · *logging* + 1
...
12. customized_elevators = move_control + full_base_functionality · **service_new** · *logging*
...

(b) A specification resulting from weaving the aspect in (a) with Specification 2

Figure 4.10: Example of using the component pointcut

the original specification by using the component pointcut. Taking Specification 2 as the base specification, the feature `logging` is introduced at the right-hand sides of Line 8 and Line 12, where `service_new` is referenced.

The equivalent_component pointcut refers to the equivalent (or indirect) appearance of the specified product families as components. Figure 4.11 exemplifies a case of using the equivalent_component pointcut to introduce a new feature `failure_capture` to Specification 1. The aspect in Fig 4.11(a) is supposed to capture any defective behaviours that are similar as the configuration `full_base_functionality`. Moreover, the definition of `full_base_functionality` cannot be changed. As shown in Specification 8 (Figure 4.11(b)), the captured join points are at right-hand sides of Line 6 and Line 9 of Specification 1.

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

(a) An aspect for modifying the reference of equivalent families

Specification 8: A PFA specification with a new feature `failure_capture` that relates to the equivalent reference of `full_base_functionality`

```
...
bf failure_capture
...
6.  base_functionality = move_control · light_display · failure_capture
                       + move_control
...
9.  customized_elevators = move_control + full_base_functionality
                         · failure_capture · service
...
```

(b) A specification resulting from weaving the aspect in (a) with Specification 1

Figure 4.11: Example of using the equivalent_component pointcut

Constraint-related pointcut

A constraint[*position_list*] pointcut is designed to capture join points associated with constraints. Three keywords, *left*, *middle* and *right*, are used to specify *position_list*, which respectively correspond to the first, second and third arguments of a PFA constraint. Example in Figure 4.12 shows how to change a requirement relation in a PFA specification using the constraint[*position_list*] pointcut. Considering that the original `service` has been changed to `service_new` by composing the aspect in Figure 4.5(a). However, the constraints related to `service_new` that are inherited from the original constraints related to `service` may become too restrictive or too loose. For example, the constraint at Line 13 in Specification 2 cannot exactly capture the relationship between `light_display` and the newly added feature `light_reset`. Therefore, an aspect (Figure 4.12(a)) is supposed to capture the reference of `service_new` at the first component of all constraints, and

to replace it by the feature `light_reset`. The result of weaving this aspect with Specification 3 is shown in Figure 4.12(b).

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

(a) An aspect for modifying the constraints

Specification 9: A PFA specification with a new constraint

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

(b) A specification resulting from weaving the aspect in (a) with Specification 2

Figure 4.12: Example of using the constraint[*position_list*] pointcut

4.3 Categories of Aspects

The previous section illustrates that a base PFA specification S is translated into a PFA specification S' by composing an aspect. We can distinguish the relationship between S and S' as augmenting, narrowing, or replacing modifications. As mentioned in Section 4.2.1, we should also distinguish the modifications by the definition and the reference of product families. Precisely, seven types of changes between S and S' can be identified as follows:

- *refine*: In S', new specification elements are added to the definition of a specified product family in S.
- *extend*: In S', new specification elements are added to some or all families of S where a specified product family is referenced.
- *discard*: The definition of a specified product family in S is absent in S'.

Table 4.2: Categories of aspects

Type of Join Points	**Effects on Join Points**	**Categories**
Definition Join Points	Augmentation	*refining*
	Narrowing	*discarding*
	Replacement	*replacing*
Reference Join Points	Augmentation	*extending*
	Narrowing	*disabling*
	Replacement	*substituting*

- *disable*: The references of a specified product family in some or all families of S is absent in S'.

- *replace*: In S', the definition of a specified product family in S is replaced by other specification elements.

- *substitute*: In S', the reference of a specified product family in some or all families of S are replaced by other specification elements.

The above classification covers all the changes a stakeholder of product families can make on the PFA specifications by composing aspects. We rely on the above categories to characterize the semantics of aspects. Each aspect should be confined to one category regarding the relationship that exists between the result PFA specification and the base PFA specification. With such categories the stakeholder can anticipate the changes imposed by composing aspects.

As we have discussed in the previous section, the changes on definition of product families and reference of product families can be distinguished by the type of the kind pointcut. Moreover, adding, removing, an replacing specification elements can be related to different modification scenarios (i.e., augmenting, narrowing, and

replacing) at the selected join points. We say an aspect *refines* the base specification if its pointcut associated with definition join points and its advice specifies an augmenting effect. The aspect in Figure 4.8 is an example of *refining* aspect, considering the new feature `service_new` *refines* the original feature `service`. An aspect *extends* the base specification if its pointcut associated with reference join points and its advice specifies an augmenting effect. The aspect in Figure 4.6 is an example of *extending* aspect, considering the feature `light_reset` is an extension to the feature `light_display`. Aspects associated with definition join points (reference join points) with replacing effects are *replacing* (*substituting*) aspects. The aspect in Figure 4.5 indicates the feature `service` is *replaced* by complete new definitions, and the aspect in Figure 4.12 indicates the reference of feature `service_new` is *substituted* by a new feature `light_display`. Similarly, aspects associated with definition join points (reference join points) with narrowing effects are referred to as *discarding* (*disabling*) aspects. In summary, given the syntax of an aspect, we category the aspect according to Table 4.2.

4.4 Usage of the Specification Language AO-PFA

We aim to handle unanticipated changes and crosscutting concerns in feature models by using the aspect-oriented paradigm. In this section, we illustrate the usage of the proposed language by using it to articulate unanticipated changes and crosscutting concerns in Sections 4.4.1 and 4.4.2, respectively.

4.4.1 Articulating Unanticipated Changes

Several examples using the proposed language are given in Section 4.2.3. A brief discussion on those examples is given below to illustrate the flexibility of the proposed language for making unanticipated changes on a PFA specification. The different effects of aspects show that our pointcut language is capable of distinguishing slight differences among requirements.

Introduction of similar requirements generating different sets of products We can find that the aspects in Figure 4.5(a) and Figure 4.12(a) together capture similar requirements as the aspect in Figure 4.6(a). The aspect in Figure 4.5(a) is for introducing the new feature `light_reset` to Specification 1 by further defining `service` (see Specification 2 in Figure 4.5(b)). Consequently, weaving the aspect in Figure 4.12(a) with Specification 2 indicates that `light_reset` requires `light_display` in all products of the elevator product family. On the other hand, the aspect in Figure 4.6(a) is for introducing the new feature `light_reset` to all products where `light_display` is available in Specification 1. In both situations, a new feature `light_reset` is introduced in the original Specification 1 and the appearance of `light_reset` indicates the appearance of `light_display` in each product. However, we should notice that the product families generated from the above two situations are not exactly the same. The former one removes the products that do not satisfy the requirements from the product family (see Specification 9 in Figure 4.12(b)), while the latter one composes a new feature to the original product families to make them all satisfy the requirements (see Specification 3 in Figure 4.6(b)).

Introduction of similar requirements at different types of locations

Aspects in Figure 4.8(a) and Figure 4.10(a) both introduce a new feature `logging`. The difference of the two aspects lies in whether or not the definitions of a specified product family are changed. The aspect in Figure 4.8(a) modifies the original definition of `service_new`. All references to the feature `service_new` in the specification, including the one in the constraint, have to be changed to the new one (see Specification 5 in Figure 4.8(b). On the other hand, the aspect in Figure 4.10(a) does not change the original definition of `service_new` and the reference to this product family in the constraint is unchanged (see Specification 7 in Figure 4.10(b)).

Introduction of similar requirements with different feature relationships

The product family algebra terms at the right-hand sides of the advice equation are the same for all aspects in Figures 4.7(a), 4.9(a), and 4.11(a), and the base specification is Specification 1 in all of the three cases. In other words, all aspects introduce a new feature `failure_capture` to the original Specification 1. However, as we can see the resulting specifications are quite different. The changes to the base specification in Figure 4.9(b) and Figure 4.11(b) are related to `full_base_functionality`. The changes of the base specification in Figure 4.7(b) are related to `move_control`. Furthermore, besides the slight difference in meaning, the main difference between the examples of Figure 4.11 and Figure 4.9 resides in whether or not the definitions of the *full_base_functionality* family (or its components) have changed.

4.4.2 Articulating Crosscutting Concerns

To validate the proposed approach for handling crosscutting concerns, we use it to specify a trial case of E-shop product families given in [BMB+10]. Regarding the base functionalities of the E-shop product families, we identify two base concerns, user interface concern and back office concern. The user interface concern focuses on a particular interest to developers of user requirements and interactions. The back office concern is a concern regarding operations of an E-shop application. In particular, `StoreFront` in the specification at the left-hand side of Figure 4.13 captures commonalities and variabilities of the back office concern. Two constraints are used to specify the "exclude" and "require" relations among features for the user interface concern. In the specification at the upper-right of Figure 4.13, `Business_Management` captures commonalities and variabilities of the back office concern. The specification at the lower-right concern of Figure 4.13 corresponds to the integration of the two views of the E-Shop product family. In particular, one additional constraint is specified regarding the view reconciliation of the two views.

We consider a crosscutting concern of security for the E-shop product families. The specification at the bottom of Figure 4.14 captures generic commonalities and variabilities of the `password` policy related to the security concern. The security concern should be reused multiple times and be composed with base concerns at multiple places. For example, we consider add `password` policy to `Administration` in the user interface concern, and replace the feature `Registration_Enforcement` in the `Registration` in the back office concern. Moreover, we impose that the option `never` in the `password` should be disabled in the first case, and the option

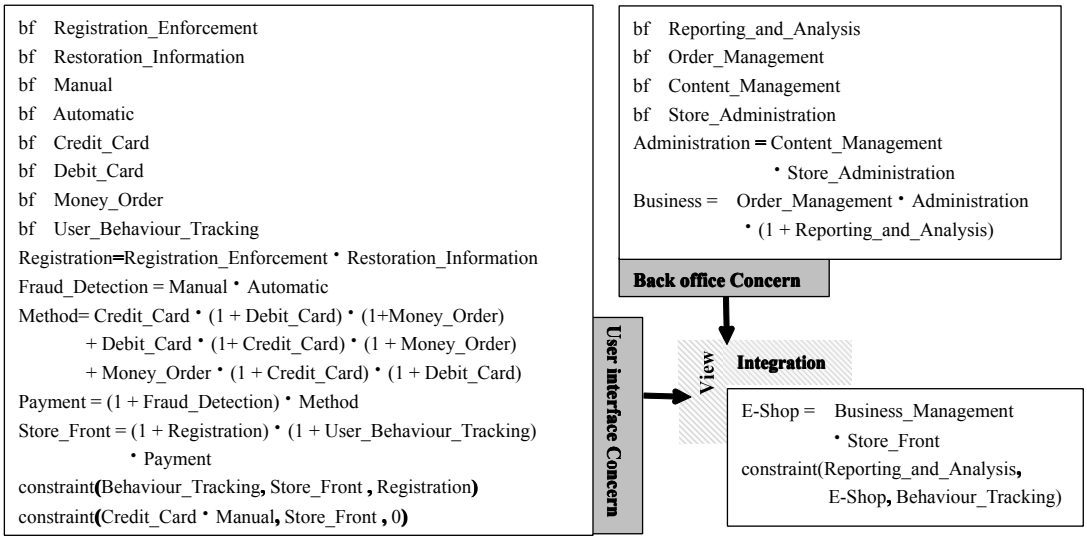

Figure 4.13: Base concerns in the E-shop feature model

`Special_Char` should be disabled in the latter case. In other words, we specify aspects for each variations of the security concern to compose the security concern with each base concerns of the E-shop product family. The specification at the upper left conner of Figure 4.14 specifies the composition of the security concern with the user interface concern, while the specification at the upper right concern of Figure 4.14 specifies the composition of the security concern with the back office concern.

The above example illustrates how the proposed techniques are used to specify feature models according to the aspect-oriented paradigm. Moreover, comparing with the technique proposed in [BMB+10], the AO-PFA specification language is more flexible in specifying the aspects and making changes to the original feature models. The AO-PFA specification language is capable of specifying most of the potential enhancement mentioned in [BMB+10].

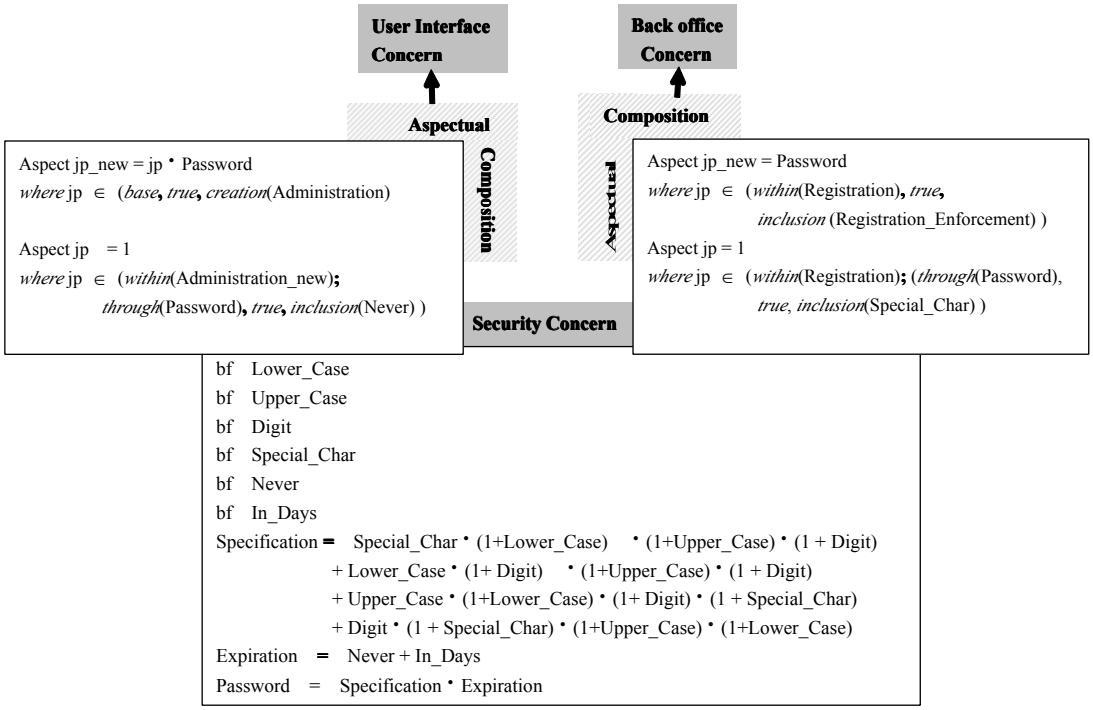

Figure 4.14: Crosscutting concerns in the E-shop feature model

4.5 Conclusion

In this chapter, I introduced the syntax of a specification language that adapts the aspect-oriented paradigm to the context of product family algebra. Moreover, a classification system based on the syntax of aspects is proposed to categorize the expected results of weaving the aspects to their base specification. The semantics of the proposed language is described with the help of a running example of an elevator product family. The running example illustrates that the proposed language enables one to modify a PFA specification in a flexible yet systematic way. I also employed the proposed aspect-oriented technique to a small case study, which illustrates the usage of the proposed language for handling crosscutting concerns at the feature-modeling level. In the following chapters, the language is used as the

basis to further enhance the verification and automation of aspectual composition at the feature-modeling level.

Chapter 5

Verifying Aspectual composition in AO-PFA

In this chapter, we focus on how to ensure the correctness of aspectual composition in AO-PFA. Section 5.1 discusses problems raised by aspectual composition and present techniques used to handle them in general. Section 5.2 presents the proposed formal technique for verifying aspectual composition in our context. Section 5.3 illustrates the usage of the proposed verification technique on a case study about home automation product family. We conclude in Section 5.4.

5.1 General Description

Aspect-oriented approaches extend the conventional notations with constructs for aspects which can be composed to base systems. Consequently, the extension of aspects raises additional correctness issues. Besides checking the correctness of an aspect, we also need to check the correctness of aspectual composition. As

mentioned in Chapter 1, obliviousness is a characteristic that can improve the extensibility and flexibility of the system. However, "total obliviousness" causes problems since the aspect may introduce unintended behaviors to the base system. The solution is to restrict the aspects that can be weaved to a base system.

To verify the correctness of aspectual composition, it is straightforward to verify properties directly on the weaved systems. However, we prefer to verify the correctness of aspectual composition prior to the weaving process. The main reason is that the former approach can be quite inefficient, especially when weaving a frequently changed aspect with a fixed base system. The verification time is proportional to the size of the weaved system, and we need to repeat the verification every time when the aspect is changed. Moreover, it is not always feasible to compose an aspect and a base system prior to the verification of aspecutal composition. Since the weaving process cannot be reversed, sometimes it is unsafe to verify the aspectual composition after the weaving process. Aspects can add malicious actions to the base system and cause severe problems. Therefore, in this chapter we focus on techniques that analyze aspects and base systems separately and check the compatibility of them before the weaving process. Moreover, regarding to the two groups of properties mentioned in Section 2.1.5, the proposed technique limits the focus on the verification of aspectual composition associated with *inheritance properties*. In spite that it is important and necessary to verify the promised properties of the aspects, the first principle for aspectual composition should be that the aspect will "Do no Harm" to the base system [Kat04].

As mentioned in Section 2.1.5, static code analysis, modeling checking, and

deductive proofs (assume–guarantee style) are commonly used for the formal verification of aspectual composition. For verifying aspectual composition in AO-PFA, we employ the static analysis technique since this technique is preferable for aspect-oriented languages with simpler syntax. In particular, the proposed verification technique is inspired by a static code analysis approach described in [RSB04] that characterizes the direct and indirect interactions of aspects with base systems. In [RSB04], the interactions of aspects are classified as orthogonal, independent, observation, actuation, and interference. By considering aspects and base systems in the context of AO-PFA, the above classification helps us to extract necessary validity criteria (i.e., definition-validity, reference-validity and dependency-validity) for PFA specifications and aspects. Although employing the static analysis technique can easily verify the preservation of properties at the syntactic level, there are other properties that cannot be established and verified with static analysis. Regarding to properties associated with the latter case, the correctness of aspectual composition remains to be verified by using other techniques, which is out the scope of this thesis.

5.2 Formal Verification of Aspectual Composition in AO-PFA

At the feature-modeling level, aspectual composition imposes amendments at specific points of a feature model, which has the potential of spoiling the validity of the original feature model. In particular, the interference of an aspect with the base specification of a feature model might alter the relationship among features

in a undesired manner. In this section, we formalize the validity criteria for PFA specifications and analyze the impacts of aspects on PFA specifications to detect invalid aspects w.r.t. a base specification.

5.2.1 Validity Criteria of PFA specification

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 finest granularity with regard to those labels.

Construction 5.1. *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 family equations. We call M_S the defining label multi-set associated with the specification S.*

Definition 5.2 (Definition-valid specification)**.** *We say that a PFA specification S is definition-valid iff $\forall(v \mid v \in M_S : NumOccur(v) = 1)$ where M_S is the defining label multi-set of S, and $NumOccur(v)$ denotes the number of occurrence of v in M_S.*

Definition 5.2 indicates that a specification is definition-valid iff all the elements in 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 D_{S}, which contains all elements of M_{S}. Correspondingly, we call D_{S} the defining label set associated with a specification S.

Construction 5.3. *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 family equations. We call R_S the referencing label set associated with the specification S.*

Definition 5.4 (Reference-valid specification)**.** *We say a specification S is reference-valid iff $R_S \subseteq M_S$, where R_S is the referencing label set of S and M_S is the defining label multi-set of S.*

Definition 5.4 indicates that a specification is reference-valid iff all the elements in R_{S} are not references to undefined labels.

Construction 5.5. *Given a PFA specification S, let D_S be its corresponding defining label set and G_S= (V, E) be a digraph. The set of vertices $V \subseteq D_S$, and 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 family equation in S. We call G_S the label dependency digraph associated with the specification S.*

Let $G_{\mathrm{S}} = (V, E)$ be a label dependency digraph associated with a PFA specification S. For $u, v \in V$, we say that u *define*s 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*, iff $\exists(m, n \mid m, n \geq 1 : (u, v)\text{–path} \in E^m \wedge (v, u)\text{–path} \in E^n))$. In particular, if u and v are identical and $m = n = 1$, we say u (or v) is *self-defined*.

Definition 5.6 (Dependency-valid specification)**.** *Let $G_S = (V, E)$ be the label dependency digraph associated with the specification S. We say that the PFA specification S is dependency-valid iff $\forall(u, v \mid u, v \in V : \neg mutdef(u, v))$ where V and $mutdef(u, v)$ are defined according to G_S.*

Definition 5.6 indicates that a valid PFA specification does not have any *mutually defined* or *self-defined* labels. Straightforwardly, the digraph G_S associated with a dependency-valid PFA specification S should be cycle-free and loop-free (The proof can be found in Appendix A.1, page 146).

5.2.2 Validity Criteria for Aspectual Composition

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

Construction 5.7. *Let S' be the resulting 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 5.1–5.5, 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 labelled family equations in S'. The defining label multi-set of S' becomes $M_{S'} = D_S \sqcup D_A$ where $\sqcup$ denotes the multi-set union. 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 family equations in S'. The referencing label set of S' becomes $R_{S'} = R_S \cup R_A$.*

- *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 family equation in S' and v is the label present at the left-hand side of the labeled family equations. 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 family equations in S, and v is any label present at the left-hand side of the labeled family equations. The dependency digraph of S' becomes $G_{S'} = (D_{S'}, (E_S \cup E\_add_A) - E\_del_A)$.*

Consequently, an aspect is invalid if it transforms a valid specification to be either definition-invalid, reference-invalid, or dependency-invalid according to Definitions 5.2, 5.4, and 5.6. Instead of checking the validity of specifications after weaving the aspects, the following subsections discussed formal techniques that detect the above validity problems before the weaving process.

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

According to the semantics of AO-PFA given in Chapter 4, the referencing label set of an aspect is always specified by the right-hand side of the advice equation (i.e.,Advice(jp)), while the defining label set of an aspect is decided by both the right-hand side and the left-hand side of the advice equation (i.e.,aspectId and Advice(jp)). In particular, aspectId defines new labels when the pointcut is associated to definition join points (i.e.,declaration, creation and component_creation), and Advice(jp) can also implicitly define new labels to the base specification. Besides, since the constraint[*position_list*] pointcut cannot constructively change the definition of any product family in the base specification, no new labels can be defined

Table 5.1: Effects of aspects with different types of kind pointcuts on D_{A} and R_{A}

kind of pointcut	$D_{\mathbf{A}}$ **contains**	$R_{\mathbf{A}}$ **contains**
declaration	All labels specifies in aspectId, and all newly introduced labels specified by Advice(jp)	All labels specified by Advice(jp)
creation		
component_creation		
inclusion	All newly introduced labels specified by Advice(jp)	The label specified by the kind pointcut and all labels specified in Advice(jp)
component		
equivalent_component		
constraint[*list*]	empty set	

by aspects with constraint[*position_list*] pointcut. Table 5.1 gives the construction of D_{A} and R_{A} with regard to different types of kind pointcut.

Definitions 5.8 and 5.9 below respectively give the formal definitions for definition-valid aspects and reference-valid aspects according to Construction 5.7. Definition 5.8 indicates that a definition-valid aspect would not lead to potentially ambiguous definitions for labels in the original specification. Definition 5.9 indicates that a reference-valid aspect would not introduce undefined references to labels in the original specification.

Definition 5.8 (Definition-valid aspect)**.** *We say that an aspect A is definition-valid w.r.t. a specification S iff $D_S \cap D_A = \emptyset$.*

Definition 5.9 (Reference-valid aspect)**.** *We say that an aspect A is reference-valid w.r.t. a PFA specification S iff $(R_S \cup R_A) \subseteq (D_S \cup D_A)$.*

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

Unlike detecting violations of definition-validity and reference-validity, detecting the violation of dependency-validity of an aspect is not straightforward. The conventional way to detect dependency-invalid aspects is to construct the dependency digraph from the weaved specifications and detect cycles and loops in the digraph. However, such an approach would be time-consuming, and the weaving process would be unnecessary if the aspect is invalid w.r.t. a base specification. In order to detect such dependency-invalid aspects prior to the weaving process, we formalize the complex relations between the given aspect specification and the dependency digraph of a base specification to verify if the aspectual composition should be allowed.

Definition 5.10 (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 5.10 indicates that a valid aspect would not lead to *mutually defined* or *self-defined* labels in the specification obtained by the weaving process. With regard to Construction 5.7, instead of examining all edges in E_add_A and E_del_A, we only consider the edges that may introduce loops or cycles in $G_{\text{S'}}$. Firstly, we use the following construction to identify vertices that are directly affected by weaving an aspect to a base specification. Instead of examining all edges in E_add_A and E_del_A, we only consider the edges that may introduce loops or cycles in $G_{\text{S'}}$. The following construction identify vertices, which are directly affected by weaving an aspect, on the label dependency digraph of a base specification,

Construction 5.11. *Given an aspect A and an base specification S, let G_S be the label dependency digraph associated with a valid specification S.*

- *We denote a vertex in G_S corresponding to the product family algebra term specified by the kind pointcut of A by k.*
- *We denote a vertex in G_S corresponding to the label specified by the scope pointcut of A by s.*

Theorem 5.1 (Kind Pointcut Condition)**.** *An aspect A is dependency-valid w.r.t. a valid PFA specification S if the type of kind pointcut of A is "*constraint*[list]".*

Proof. When the kind pointcut is constraint[*list*], the aspect A only affects the product families within the constraints. According to Construction 5.7, it indicates that $D_{\mathrm{A}} = \mathrm{E\_add_A} = \mathrm{E\_del_A} = \emptyset$. We accomplish the proof by proving ($D_{\mathrm{A}} = \mathrm{E\_add_A} = \mathrm{E\_del_A} = \emptyset \implies A$ is dependency-valid w.r.t. S). The detailed proof is provided in Appendix A.1, page 147. □

Theorem 5.2 (Non-cycle Condition)**.** *Let S be a valid PFA specification and A be an aspect that does not satisfy the kind pointcut condition (Theorem 5.1). Construct the label dependency digraph G_S according to Construction 5.5 and denote or create the vertex k in G_S according to Construction 5.11. 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. In Theorem 5.2, the function $Walk : V \times V \to ordered - list(V)$ is defined over a digraph such that $Walk(u, v)$ returns the list of all vertices along a walk from u to v. 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. 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. The proof of this proposition is provided in Appendix A.1, page 148. □

From the proof of Theorem 5.2, we have the following equation:

$$\begin{aligned} & A \text{ is dependency-invalid w.r.t. } S \\ \Longleftrightarrow \quad & \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B}_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{B}_{\mathrm{def}} : \exists(m \mid m \geq 1 : \\ & \qquad\qquad (u, v)\text{–path} \in (E_S)^m)\,)\big) \end{aligned} \tag{5.1}$$

where $\mathrm{B}_{\mathrm{def}}$ is the set of labels at the left-hand sides of a set of labeled family equations where join points are present at their right-hand sides.

Definition 5.12. *We say that an aspect is potentially dependency-invalid if it does not satisfy both the point kind condition (Theorem 5.1) and the non-cycle condition (Theorem 5.2).*

Definition 5.12 indicates that an aspect is possibly dependency-invalid iff its kind of pointcut is not constraint[*list*] and $\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : Walk(k, v) \neq \emptyset\,))$. Using the above properties, we verify whether an aspect is potentially invalid in accordance to the type of scope 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.

Lemma 5.3. *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. When the scope of the pointcut is "*protect*(base)", A is always dependency-valid w.r.t. S.*

Proof. We use Equation 5.1 and substitute the definition of B_{def}. When the type of the scope pointcut is *base*, join points are where k is present. Therefore, $B_{def} = N^+(k)$, where $N^+(k)$ denotes the set of all successors of the vertex k. The detailed proof is provided in A.1, page 151. When the type of the scope pointcut is protect(*base)*, the set B_{def} is empty. In the proof, we use the definition of $Walk(u, v)$, path concatenation, the one-point rule, range split, empty range and ∃-false body. The detailed proof is provided in A.1, page 152. □

Lemma 5.4. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 5.5 and denote or create the vertex k in G_S according Construction 5.11.*

- *When the type of the scope pointcut is "*within*", A is dependency-invalid w.r.t. S iff* $\exists(v \mid v \in D_S \cap R_A : s \in Walk(k, v) \wedge s \neq k)$.
- *When the type of the scope pointcut is "*protect*(*within*)", A is dependency-invalid w.r.t. S iff* $\exists(v \mid v \in D_S \cap R_A : s \notin Walk(k, v) \vee s = k)$.
- *When the type of the scope pointcut is "*through*", A is dependency-invalid w.r.t. S iff* $\exists(v \mid v \in D_S \cap R_A : s \in Walk(k, v) \wedge s \neq k \wedge s \neq v)$.
- *When the type of the scope pointcut is "*protect*(*through*)", A is dependency-invalid w.r.t. S iff* $\exists(v \mid v \in D_S \cap R_A : s \notin Walk(k, v) \vee s = k \vee s = v)$.

Proof. Equation 5.1 and Lemma 5.3 are used in the proof. When the type of the scope pointcut is within, join points are bound to a labeled family equation whose label is s. Therefore, we have $B_{def} = \{s\}$. When the type of the scope pointcut is through, join points are bound to a labeled family equation where s is present at the

right-hand side. Therefore, $B_{def} = N^+(s)$. Besides, there should be a path from k to s. On the other hand, when the type of the scope pointcut is protect(within), labels in B_{def} should not include s if there is a path from k to s. When the type of the scope pointcut is protect(within), labels in B_{def} should not include successors of s if there is a path from k to s. Otherwise, the set B_{def} is identical to the one specified by a pointcut with scope *base*. The detailed proof is provided in A.1, page 152–157. □

Lemma 5.5. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. When the scope pointcut is the union of two scopes of types ts_1 and ts_2, A is dependency-invalid when A is dependency-invalid w.r.t. at least one of the scopes. When the scope pointcut of A is the intersection of two scopes of types ts_1 and ts_2, A is dependency-invalid when A is both dependency-invalid w.r.t. the two scopes.*

Proof. Let the set of join points selected by the scope pointcut of type ts_1 be B^1_{def} and the set of join points selected by the scope pointcut of type ts_2 be B^2_{def}. When the type of the scope pointcut is $(ts_1 : ts_2)$, then the set $B_{def} = B^1_{def} \cup B^2_{def}$. When the type of the scope pointcut is $(ts_1 \ ; \ ts_2)$, then the set $B_{def} = B^1_{def} \cap B^2_{def}$. Provided that $B_{def} \neq \emptyset$, we substitute the definition of B_{def} in Equation 5.1 and use Lemmas 5.3 and 5.4 to accomplish the whole proof. □

Theorem 5.6 (Dependency-invalid aspect)**.** *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Denote the vertex that invalidates the condition of Theorem 5.2 by a. Vertices k and s are denoted or created in G_S as prescribed in Construction 5.11. Let Dep_invalid(ts) be the following predicate,*

where ts represents the type of the scope pointcut.

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

Provided the set of join points is nonempty, the aspect A is dependency-invalid w.r.t. S if Dep_invalid(ts).

Proof. The proof of the above proposition is based on Lemmas 5.3, 5.4, and 5.5. The complete detailed proof is provided in A.1, page 159. □

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.

5.3 Usage of the verification technique

The verification technique can be connected with the classification of aspects discussed in Section 4.3. Intuitively, *extending*, *disabling*, and *substituting* aspects are always definition-valid according to Definition 5.8. Since $D_A \subseteq R_A - D_S$, or $D_A = \emptyset$, we always have $D_S \cap D_A = \emptyset$. According to Definition 5.9, *refining*, *discarding*, and *replacing* aspects are always reference-valid w.r.t. a reference-valid base specification. Since $R_A - D_S \subseteq D_A \iff R_A \subseteq D_A \cup D_S$. Assume $R_S \subseteq D_S$, we have $R_S \cup R_A \subseteq D_A \cup D_S$. Moreover, since $R_A = \emptyset$ for *discard* and *disable* aspects, which indicates *discard* and *disable* aspects are always dependency-valid according to Theorem 5.2.

Moreover, features are the main abstraction mechanism of feature-oriented software development. The synthesis of a concrete product from feature composition is a challenge work. First of all, not all features specified in a product family are compatible. Besides domain constraints, there are invalid feature compositions that are caused by the unintended interactions of features. The safe feature composition problem has been widely studied in the area of feature-oriented software development, and is challenging to tackle [TBKC07, KBK09]. Conventionally, the interactions of features are analyzed at the design and implementation levels for features (See Stage 2 and Stage 3 in Figure 1.1). With the proposed aspect-oriented technique, the feature composition can be analyzed at a more abstract level by aspectual composition in feature models. Consequently, in our context, the analysis of feature dependency and interactions is translated into the aspectual dependencies and interaction problems at the feature-modeling level. Instead of detecting

Table 5.2: Abbreviation of basic features and families

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		

all invalid compositions at the granularity of features, some conflicts among features are detected at an early stage as invalid aspectual composition in feature modeling. The verification technique proposed in this chapter for safe aspectual composition can be consider as a technique that verifies the safe composition of features from an early stage. Consequently, we can make necessary trade-off as early as possible at the feature-modeling stage, and provide valuable knowledge for the following design and implementation stages for developing product families. In this section, several cases of invalid aspectual composition are studied to illustrate how the proposed techniques can help to identify conflicts for feature compositions and make early trade-off at the feature-modeling level.

5.3.1 Case Study: Home Automation Family

A home automation system [PBvdL05] 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. We only use

Specification : Home_Automation_Product_Line

1. bf tim_con	18.Lig_dev = 1+lum_sen
2. bf lum_sen	19.Daw_dev = tim_con· (1+wea_sen)
3. bf wea_sen	20.Usr_Int = gra_tv· (1+web_bas) · (1+PDA)
4. bf gra_tv	21.Commun = (cable+wireless+cable· wireless)
5. bf web_bas	· (1+inter) · (1+mob_net)
6. bf PDA	22.Lig_con = man_lig· (1+sma_lig)
7. bf cable	23.Daw_con = man_daw· (1+sma_daw)
8. bf wireless	24.HApp_con = (1+heat_sys)· (1+ster_sys)
9. bf inter	· (1+wat_intr)
10. bf mob_net	25.fir_det_flow = sma_lig· sma_daw· HApp_con
11. bf man_lig	26.Home_gateway = Lig_con· Daw_con· HApp_con
12. bf sma_lig	· fir_det_flow
13. bf man_daw	27.Home_Auto_PL = Commun· Usr_Int· Lig_dev
14. bf sma_daw	· Daw_dev· Home_gateway
15. bf heat_sys	28.constraint(sma_lig, Home_Auto_PL, Lig_dev)
16. bf ster_sys	29.constraint(sma_daw, Home_Auto_PL, Daw_dev)
17. bf wat_intr	30.constraint(web_bas, Home_Auto_PL, inter)

Figure 5.1: Example of a base specification for the home automation product line

parts of the case study to show the usage of the proposed verification technique for aspectual composition. We use the abbreviations shown in Table 5.2 to shorten expressions, and Figure 5.1 is a PFA specification corresponding to the feature model of a home automation product line.

Aspects causing definition-invalid specifications

We consider the following two aspects that are developed independently. The aspect denoted as `case_5.1a` intends to deploy a fingerprint reader device as an optional feature in `Daw_dev`, while the aspect denoted as `case_5.1b` intends to deploy the fingerprint reader device as a mandatory feature. We denote the fingerprint reader by `fgr`. The two aspects can be respectively specified in AO-PFA as follows:

case_5.1a:	
Aspect	jp_new = jp · (1 + fgr)
where	jp ∈ (base, true, creation(Daw_dev))

case_5.1b:	
Aspect	jp_new = jp · fgr
where	jp ∈ (base, true, creation(Daw_dev))

Let the base specification associated with the above two aspects be the specification *home_automation* (Figure 5.1). Both aspects (case_5.1a and case_5.1b) will capture the left-hand side of the labeled family `Daw_dev` at Line 19 in Figure 5.1 and introduce a new labeled product family `Daw_dev_new`. In other words, weaving these two aspects to the same base specification can result in a definition-invalid PFA specification. To detect such conflict formally, we assume that, without loss of generality, the aspect case_5.1a is weaved before the aspect case_5.1b. Using Construction 5.1, the defining label multi-set of the base Specification *home_automation* is as follows:

$M_{\text{home\_automation\_product\_line}}$={ 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_daw, heat_sys, wat_intr, Lig_dev, Daw_dev, fir_det_flow, Commun, Lig_con, HApp_con, Usr_Int, Home_gateway, Daw_con, Home_Auto_PL}.

With regard to Definition 5.2, we claim that the Specification *home_automation* is definition-valid. We then have $D_{\text{home\_automation\_product\_line}} = M_{\text{home\_automation\_product\_line}}$. We construct the defining label set of the aspects according to the row corresponding to creation in Table 5.1 and obtain $D_{\text{case\_5.1a}} = D_{\text{case\_5.1b}} = \{fgr, Daw\_dev\_new\}$. According to Definition 5.8, since the intersection of $D_{\text{home\_automation}}$ and $D_{\text{case\_5.1a}}$ is empty, the aspect case_5.1a is definition-valid w.r.t. its base specification. Let *home_automation_one* be the specification after weaving the aspect case_5.1a. According to Construction 5.7, we have $D_{\text{home\_automation\_one}} = D_{\text{home\_automation}} \sqcup D_{\text{case\_5.1a}}$. As the intersection of $D_{\text{home\_automation\_one}}$ and $D_{\text{case\_5.1b}}$ is nonempty, the aspect case_5.1b is definition-invalid w.r.t. its base specification *home_automation_one*. The definition-invalid aspectual composition identifies the conflicts between two

independent aspects, which both intend to modify the definition of another features. Consequently, trade-offs between feature interactions are enabled at an earlier stage by solving such conflicts of the two aspects at the feature-modeling level.

Aspects causing reference-invalid specifications

We consider another aspect (denoted as case_5.2) that extends the fingerprinter reader `fgr`, with a new feature `password_identify`. The aspect is specified as follows:

case_5.2:	
Aspect	jp = jp · password_identify
where	jp ∈ (base, true, inclusion(fgr))

The inclusion pointcut refers to join points at the reference of the feature `fgr`. Assume that we want to weave both aspects case_5.1a and case_5.2 to the base specification *home_automation* (Figure 5.1). If we weave the aspect case_5.2 first, there is actually no modification to the base specification after the weaving process as the feature `fgr` does not appear in the base specification. The feature `fgr` then appears in the new specification after weaving the aspect case_5.1a. However, the aspect case_5.2 does not take effect, which may not correspond to our expectation. Actually, weaving the first aspect leads to a reference-invalid specification. Intuitively, it is more reasonable to weave the aspect case_5.1a before the aspect case_5.2. To formally detect the above problem with the proposed verification technique, we construct the defining label set and referencing label set of the aspect case_5.2 according to the row of inclusion in Table 5.1. We obtain $D_{\text{case\_5.2}} = \{password\_identify\}$, $R_{\text{case\_5.2}} = \{fgr, password\_identify\}$. Based

on Definition 5.9, the aspect case_5.2 is reference-invalid w.r.t. the specification *home_automation*. On the other hand, suppose that we weave the aspect case_5.1a before the aspect case_5.2. Thus, *home_automation_one* is the base specification associated with the aspect case_5.2. According to Table 5.1 and Construction 5.7, $R_{\text{home\_automation\_one}} = R_{\text{home\_automation}} \cup \{fgr\}$. Consequently, the aspect case_5.2 becomes reference-valid w.r.t. *home_automation_one*. The reference-invalid aspectual composition is indeed caused by introducing features reference to undefined features. As illustrated in the above example, the detection of such invalid aspectual composition at the feature-modeling level helps us to derive an correct order for composing aspects. Consequently, the order in which features are composed is restricted by the composition order of aspects.

Aspects causing dependency-invalid specifications

Considering the composition of base specification *home_automation* (Figure 5.1) and the aspect specified below (denoted as case_5.3):

case_5.3:	
Aspect	jp = jp · fir_det_flow
where	jp ∈ (base, true, inclusion(sma_lig))

The dependency digraph of the specification *home_automation* is given in Figure 5.2. The digraph is loop- and cycle-free, which indicates that the corresponding specification is dependency-valid. The dotted edge illustrates the new edge introduced by the aspect case_5.3, which introduces a loop to the dependency digraph. The example shows that weaving an aspect might lead to a dependency-invalid specification. With accordance to the proposed detection technique, we can confirm the above results without explicitly constructing new edges that are introduced

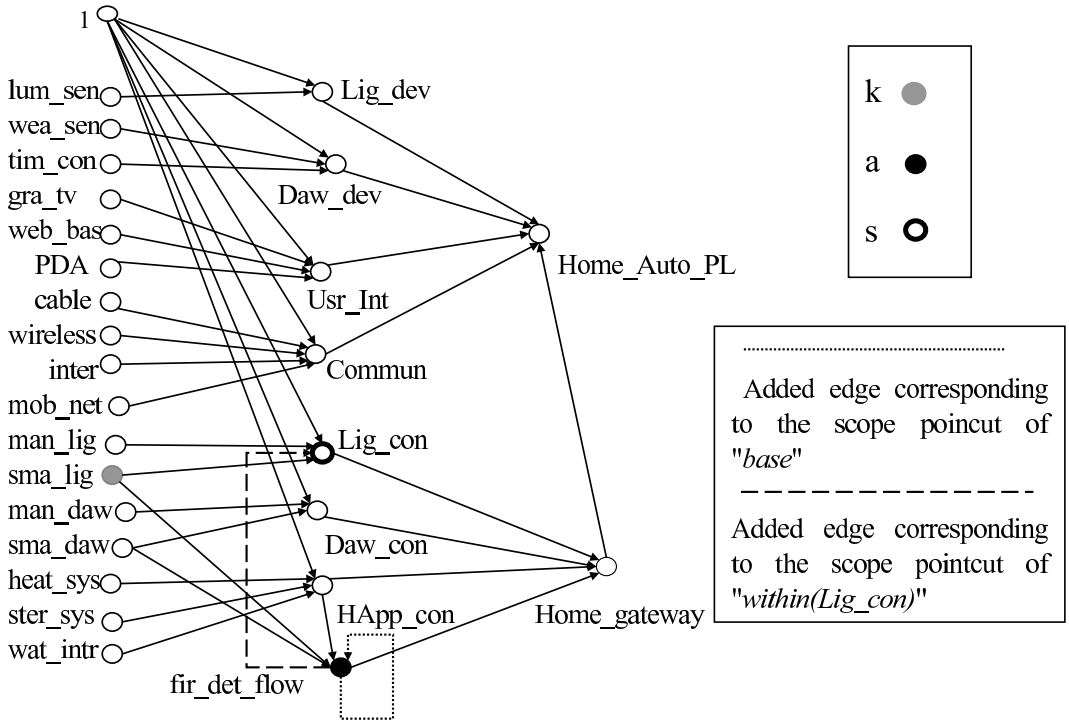

Figure 5.2: Dependency digraph corresponding to the home automation product line

by aspects. We represent vertices that are in both the referencing label set of the aspect and the defining label set of the base specification by black vertices in the label dependency digraph. The vertices k and s are represented by the gray vertex and bold circle vertex, respectively. Based on Theorem 5.2 and the first item in Theorem 5.6, the aspect case_5.3 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`. Alternatively, assume that we change the scope of the above pointcut to be within(Lig_con) instead of *base*, the dashed edge introduced by the modified aspect will not cause loops or cycles in the dependency digraph. Formally, the modified aspect with a bounded scope is dependency-valid w.r.t. its base specification based on the second item in Theorem 5.6. The above aspects illustrate an example of how to avoid

dependency-invalid aspects by imposing additional information to the aspectual composition; the aspect should be composed to the base specification within a certain scope. The restriction on aspectual composition can be used for the analysis of safe feature composition and further mapped to other development stages that follow the feature-modeling.

5.4 Conclusion

In this chapter, I presented a formal technique to verify aspectual composition in AO-PFA. A set of definitions, constructions, and propositions are proposed and used to check the compatibility of aspects w.r.t. to their base specifications. In the literature of requirement engineering and architectural design, most aspect-oriented approaches are informal, and their support of validation and verification is limited. The verification of aspectual composition in those approaches is only accomplished by informally "walking through" the artifacts [batmaccRS+05]. Compared with conventional approaches, the specification language AO-PFA presented in Chapter 4 provides a formal and compact way to modularize and compose aspects and base specifications. Moreover, the verification of aspectual composition is prior to the weaving of aspects and base specification. We will discuss the weaving process in the next chapter.

Chapter 6

Weaving Aspects in AO-PFA

In this chapter, we focus on the issue about weaving aspects in AO-PFA. Section 6.1 gives formal semantics for a proposed weaving process. Section 6.2 discusses several theoretical properties related to the weaving process. In Section 6.3, we conclude.

6.1 Semantics of the Weaving Process

A weaver is used to automate the weaving process for an aspect-oriented language. Base specifications and aspects are the input of the weaver. One of the key issues in defining the formal semantics of the weaving process is establishing the formal representations for both base specifications and aspects. A second issue is formalizing the exact behaviours of the weaver. In the remainder of this section, the two key issues above are further discussed in the context of AO-PFA.

6.1.1 Formalism of PFA Specifications and Aspects

We recall that PFA specifications are based on product family algebra, which is a commutative idempotent semiring. Moreover, the PFA language includes equations defining families and constraints on families. The semantics of PFA language can be considered as a parametrised algebraic specification ($\text{SPEC}_{\text{PFA}}, \text{SR}$). The parameter specification SPEC_{PFA} contains a sort $\mathtt{f}$, and operations $\oplus$, $\odot$, ⓪ and ① . The interpretations of $\mathtt{f}$, $\oplus$, $\odot$, ⓪ and ① respectively corresponds to the interpretations of f, $+$, $\cdot$, 0, and 1 as given in Definition 3.5[1]. The sort $\mathtt{f}$ denotes the sort of product families, $\oplus$ and $\odot$ correspond to the choice and mandatory composition of product families, ⓪ denotes the empty product family, and ① denotes the pseudo-product. The body specification SR contains a set of axioms corresponding to the commutative idempotent semiring, which is denoted as E_f. The set E_f is given in Table 6.1, where each axiom is represented according to Definition 3.14. For instance, $\big(\{x, y, z\}, (x \oplus y) \oplus z, x \oplus (y \oplus z)\big)$ represents an equation $(x \oplus y) \oplus z = x \oplus (y \oplus z)$. The corresponding target specification is denoted as follows:

$$\text{SR}[\text{SPEC}_{\text{PFA}}] = \text{SPEC}_{\text{PFA}} \uplus (\emptyset, \emptyset, E_f) = (\{\mathtt{f}\}, \{\oplus, \odot, ⓪, ① \}, E_f). \tag{6.1}$$

Let $\mathcal{F}$= $(\{f\},\ \{+, \cdot, 1, 0\})$ denote the signature of general PFA specifications,

[1]We use $\mathtt{f}$, $\oplus$, $\odot$, ⓪, and ① instead of f, $+$, $\cdot$, 1 and 0, to avoid the sharing symbols between the target specification and the actual parameter specification

Table 6.1: *Equational theory E_f*

1.	Associativity of $\oplus$:	$\big(\{x,y,z\}, (x \oplus y) \oplus z, x \oplus (y \oplus z)\big)$;
2.	Symmetry of $\oplus$:	$\big(\{x,y\}, x \oplus y, y \oplus x\big)$;
3.	Identity of $\oplus$:	$\big(\{x\}, x \oplus ⓪, x\big)$;
4.	Idempotent of $\oplus$:	$\big(\{x\}, x \oplus x, x\big)$;
5.	Associativity of $\odot$:	$\big(\{x,y,z\}, (x \odot y) \odot z, x \odot (y \odot z)\big)$;
6.	Symmetry of $\odot$:	$\big(\{x,y\}, x \odot y, y \odot x\big)$;
7.	Identity of $\odot$:	$\big(\{x\}, x \odot ①\ , x\big)$;
8.	Distributivity:	$\big(\{x,y,z\}, x \odot (y \oplus z), x \odot y \oplus x \odot z\ \big)$;
9.	Annihilator:	$\big(\{x\}, x \odot ⓪, ⓪\big)$;

and $\mathfrak{L}(S)$ denote the set of labels which are specified in a particular PFA specification S. Then the specification S can be considered as a $\mathcal{F}$-term algebra (Definition 3.13) generated by $\mathfrak{L}(S)$. In particular, product family algebra terms associated with the specification S are denoted by $T_{\mathcal{F}}(\mathfrak{L}(S))$. Furthermore, let Γ denote the alphabet of the PFA language and $\mathcal{M}$ be the signature of a monoid. The set $V = T_{\mathcal{M}}(\Gamma)$ represents all possible labels for features and families in PFA language specifications, which indicates $\mathfrak{L}(S) \subseteq V$. Besides, let $Eq(S)$ and $Cp(S)$ respectively denote the set of labelled family equations and the set of constraints that are specified in S. Each labelled family equation (e.g., $l = r$) is represented by a triple (X, l, r) in $Eq(S)$ such that $l, r \in T_{\mathcal{F}}(X)$, and $X \subseteq \mathfrak{L}(S)$. Each constraint (e.g., $l \xrightarrow{m} r$) is represented by a triple $(l,\ m,\ r)$ in $Cp(S)$ such that $l, m, r \in T_{\mathcal{F}}(\mathfrak{L}(S))$. A particular PFA specification S can be represented by an algebraic specification with constraints (see Definition 3.17) as follows:

$$\mathcal{S} = \big(\{f\},\ \{+, \cdot, 1, 0\} \cup \mathfrak{L}(S),\ Eq(S) \cup Cp(S)\big)\ . \tag{6.2}$$

According to Equations (6.1) and 6.2, the PFA specification S is formally represented by actualizing SPEC_{PFA} with $\mathcal{S}$, which can be written as follows:

$$\text{SR}[\text{SPEC}_{\text{PFA}} \mapsto \mathcal{S}] = \mathcal{S} \uplus (\emptyset, \emptyset, E_f) \tag{6.3}$$

Therefore, the semantics of particular PFA specifications are those of $\mathcal{S} \uplus (\emptyset, \emptyset, E_f)$, which are still semiring-based semantics. The models of PFA specifications can be trivial extensions to models of commutative idempotent semirings such as the set-based or the bag-based ones discussed in [HKM06].

As discussed in Chapter 4, an aspect in AO-PFA is composed of an advice equation $\langle \mathsf{aspectId} \rangle = \langle\ \mathsf{Advice(jp)} \rangle$, and a pointcut triple ($\langle \mathsf{scope} \rangle$, $\langle \mathsf{expression} \rangle$, $\langle \mathsf{kind} \rangle$). According to the mathematical setting given previously, we have $\langle \mathsf{aspectId} \rangle \in T_{\mathcal{M}}(\{jp\} \cup \Gamma)$, and $\langle\ \mathsf{Advice(jp)} \rangle \in T_{\mathcal{F}}(V \cup \{jp\})$. Besides, the pointcut triple is used to select a set of join points, which can be considered as a quantification over the base specification. Formally, we can interpret the pointcut of an aspect in AO-PFA as follows:

$$\forall(p \ \mid \ p \in \text{JP} \ : \ \mathit{scope}(p) \wedge \mathit{expression}(p) \wedge \mathit{kind}(p)\ ) \tag{6.4}$$

where JP is the set of all potential join points in the base specifications, and $\mathit{scope}(p)$, $\mathit{expression}(p)$, and $\mathit{kind}(p)$ are three predicates respectively associated with the three components of the pointcut triple. A join point in JP is selected by an aspect when $\mathit{scope}(p)$, $\mathit{expression}(p)$, and $\mathit{kind}(p)$ are all evaluated to be true. The precise definitions of JP and the three predicates are further discussed in the next subsection when formalizing the behaviours of the weaver for AO-PFA.

6.1.2 Formalism of the Weaver

A general process of waving an aspect to a base specification involves the following two sequential actions:

(1) Each potential join point in the given base specification should be evaluated against the pointcut of the given aspect.

(2) The advice of an aspect should be instantiated to bind with and finally introduced at each selected join point.

As identified in [BMN+06], *join point model*, *poincut matching*, *advice binding*, and *weaving execution* are four essential elements of a weaver. We respectively discuss those essential elements in our context to formalize the behaviour of the weaver for AO-PFA.

Join Points Evaluation

As mentioned earlier in Section 6.1.1, a PFA specification is composed of a set of product family terms. Roughly speaking, the join point model in AO-PFA is defined by subterms of product family algebra terms. In particular, a subterm relation $\preceq$ in our context is a reflexive and transitive relation on $T_{\mathcal{F}}(X)$ such that $\forall(t, t_1, t_2 \in T_{\mathcal{F}}(X) \mid t \preceq t_1 \vee t \preceq t_2 : t \preceq t_1 + t_2)$, and $\forall(t, t_1, t_2 \in T_{\mathcal{F}}(X) \mid t \preceq t_1 \vee t \preceq t_2 : t \preceq t_1 \cdot t_2)$. However, a term may have many subterms and different subterms of the given term may further have common subterms. We say two subterms of a term are overlapping if they have common subterms. The subterms identified as join points in a given product family algebra term should not be overlapped. Therefore, the join point model in our context is restricted to subterms

that are either labels (which are minimal subterms) in the term concerned, or the entire term itself. The join point model of AO-PFA, captured by the set JP, is formally defined according to Equations (6.5)–(6.7). Precisely, we have JP = $\mathcal{DJ}$ $\cup$ $\mathcal{E}$ $\cup$ $\mathcal{C}$. As discussed in Chapter 4, join points are distinguished as definition join points and reference join points. The set of definition join points is denoted by $\mathcal{DJ}$, which coincides with the set of free constants $\mathfrak{L}(S)$. The set of reference join points, denoted by $\mathcal{RJ}$, is the union of the sets $\mathcal{E}$ and $\mathcal{C}$.

$$\mathcal{DJ} = \{p \mid \exists(t \mid t \in \mathfrak{L}(S) : p \preceq t)\} = \mathfrak{L}(S), \tag{6.5}$$

$$\mathcal{E} = \{p \mid \exists\big(r \mid (X,\ l,\ r) \in Eq(S) : (p \preceq r \wedge p \in \mathfrak{L}(S)) \vee p = r\big)\}, \tag{6.6}$$

$$\mathcal{C} = \{p \mid \exists\big(f, s, t \mid (f, s, t) \in Cp(S) : [(p \preceq f \vee p \preceq s \vee p \preceq t) \wedge p \in \mathfrak{L}(S)] \vee (p = f \vee p = s \vee p = t)\big)\}\ . \tag{6.7}$$

Moreover, let $\mathcal{P}(\mathfrak{L}(S))$ denote the powerset of $\mathfrak{L}(S)$. We use $context_{\mathcal{DJ}} : \text{JP} \rightarrow \mathcal{P}(\mathfrak{L}(S))$ and $context_{\mathcal{RJ}} : \text{JP} \rightarrow \mathcal{P}(\mathfrak{L}(S))$ to obtain the context information for join points. Functions $context_{\mathcal{DJ}}$ and $context_{\mathcal{RJ}}$ are only defined for join points associated with labelled family equations. Let p be a potential join point in a labelled family equation, e.g., $(X,\ l,\ r) \in Eq(S)$, we have:

$$\begin{cases} context_{\mathcal{DJ}}(p) \stackrel{\text{def}}{=} (X - \{l\}) & \text{when } p \in \mathcal{DJ} \\ context_{\mathcal{RJ}}(p) \stackrel{\text{def}}{=} (\{l\}) & \text{when } p \in \mathcal{RJ} \end{cases}\ . \tag{6.8}$$

Pointcut Matching

As mentioned earlier, the matching of pointcut is to evaluate the pointcut predicates at each potential join point in the join point mode JP. The ⟨scope⟩ expression specifies the bounds of join points. A function *matchscope* : SCOPE_EXP × JP → 𝔹 is defined to match join points in a base specification (e.g., S), where SCOPE_EXP denotes the domain of valid scope expressions as defined in Figure 4.4. The semantics of union (:), intersection (;), and protect of scopes are straightforward. Moreover, the scope is always matched for all join points if the expression ⟨scope⟩ is specified as *base*. For a within scope, a potential join point is matched if the context information of a reference join point is identical with the specified scope. The matching of a through scope is somewhat complicated. A function $context\_list(L, E)$ is used to help the matching of a through scope. The function returns a set of labels, and the types of the arguments L and E are a set of labels and a set of equation. In particular, $context\_list(\{v\}, Eq(S))$ collects the labels that are along the hierarchical flow of a label v in a specification S. An auxiliary function $\mathrm{H}\big(L, (X, l, r)\big)$ w.r.t. each equation (X, l, r) is defined below to check whether the label l should be added to the list.

$$\mathrm{H}: \ \mathcal{P}(\mathfrak{L}(S)) \times Eq(S) \to \mathcal{P}(\mathfrak{L}(S)),$$

$$\mathrm{H}\big(L, (X, l, r)\big) = \begin{cases} \{l\} & \text{when } \exists\big(x \ |: \ x \in (context_{\mathcal{DJ}}(l) \ \cap \ L \ \cap \ \overline{weaved\_list})\big) \\ \emptyset & \text{otherwise} \end{cases} \tag{6.9}$$

where *weaved_list* represents a set of labels which already have at least one actual matched join point at their right-hand side. Then *context_list* is inductively defined

as follows:

$$context\_list:\ \mathcal{P}(\mathfrak{L}(S)) \times \mathcal{P}(Eq(S)) \to \mathcal{P}(\mathfrak{L}(S)),$$

$$\begin{cases} \text{base case:} & context\_list(L, \emptyset) = & L, \\ \text{inductive case:} & context\_list(L, E) = & context\_list(L \cup H\big(L, (X, l, r)\big), E - \{(X, l, r)\}). \end{cases} \tag{6.10}$$

Finally, let $v \in \mathfrak{L}(S)$, $p \in \text{JP}$, and $\text{s\_exp}_1, \text{s\_exp}_2, \text{s\_exp} \in SCOPE\_EXP$. The function *matchscope* is defined inductively as follows:

$$\begin{cases} \textbf{base cases:} & & \\ matchscope(base, p) & \overset{\text{def}}{\Longleftrightarrow} & \mathsf{true} \\ matchscope(\mathsf{within}\ v, p) & \overset{\text{def}}{\Longleftrightarrow} & (\ context_{\mathcal{RJ}}(p) = \{v\}\ ) \\ matchscope(\mathsf{through}\ v, p) & \overset{\text{def}}{\Longleftrightarrow} & (\ p \in context\_list(\ \{v\},\ Eq(S)\ )\ ) \\ \textbf{inductive cases:} & & \\ matchscope(\text{s\_exp}_1; \text{s\_exp}_2, p) & \overset{\text{def}}{\Longleftrightarrow} & (\ matchscope(\text{s\_exp}_1, p) \wedge matchscope(\text{s\_exp}_2, p)\ ) \\ matchscope(\text{s\_exp}_1 : \text{s\_exp}_2, p) & \overset{\text{def}}{\Longleftrightarrow} & (\ matchscope(\text{s\_exp}_1, p) \vee matchscope(\text{s\_exp}_2, p)\ ) \\ matchscope(\mathsf{protect}(\text{s\_exp}), p) & \overset{\text{def}}{\Longleftrightarrow} & (\ \neg matchscope(\text{s\_exp}, p)\ ) \end{cases} \tag{6.11}$$

Example 6.1. *We revisit several examples that are given in Chapter 4. The* base specification *in Figure 6.1 is Specification* 1 *introduced on page 66, and the aspect* case_6.1 *is the aspect introduced on page 75. We now illustrate how the pointcut expression can be matched as expected according to the above proposed functions. As an example, we evaluate matchscope() at the term* `full_base_functionality` *at Line* 9*, which is recognised as a potential join points according to Equation (6.6).*

Specification : Base Specification

1. bf move_control
2. bf light_display
3. bf service
4. optional_light_display = light_display +1
5. optional_service = service +1
6. base_functionality = move_control · optional_light_display
7. elevator_product_line = base_functionality · optional_service
8. full_base_functionality = move_control · light_display
9. customised_elevators = move_control +full_base_functionality · service
10. constraint(service, elevator_product_line, light_display)

case_6.1:

Aspect	jp = jp · failure_capture
where	jp ∈ (within(customised_elevators) ; through(full_base_functionality), $\#(jp) \geq 1$, inclusion(move_control))

Figure 6.1: Example to illustrate the proposed weaving process for AO-PFA

$$matchscope\big(\mathsf{within}(\mathit{customised\_elevators})\,;\mathsf{through}(\mathit{full\_base\_functionality}),\ \mathit{full\_base\_functionality}\big)$$

$\Longleftrightarrow$ $\langle$ *inductive cases in Expression (6.11)* $\rangle$

$$matchscope\big(\mathsf{within}(\mathit{customised\_elevators}),\ \mathit{full\_base\_functionality}\big) \wedge matchscope\big(\mathsf{through}(\mathit{full\_base\_functionality}),\ \mathit{full\_base\_functionality}\big)$$

$\Longleftrightarrow$ $\langle$ *base cases in Expression (6.11)* $\rangle$

$$context_{\mathcal{RJ}}(\mathit{full\_base\_functionality}) = \{\mathit{customised\_elevators}\} \wedge \mathit{full\_base\_functionality} \in context\_list(\{\mathit{move\_control}\},\ Eq(\mathsf{base\_specification}))$$

$\Longleftrightarrow$ $\langle$ *Equation (6.8)* $\rangle$

$$\mathsf{true} \wedge \mathit{full\_base\_functionality} \in context\_list(\{\mathit{move\_control}\},\ Eq(\mathsf{base\_specification}))$$

$\Longleftrightarrow$ $\langle$ *Equation (6.9)* & *Equation (6.10)* $\rangle$

$$\mathsf{true} \wedge \mathit{full\_base\_functionality} \in \{\mathit{move\_control}, \mathit{base\_functionality}, \mathit{full\_base\_functionality}\}$$

$\Longleftrightarrow$ $\langle$ *set and logic axioms* $\rangle$

true

The ⟨kind⟩ expression specifies the extract positions of join points. In the context of algebraic specifications, the problem is to extract subterms that are equivalent to a specified product family algebra term w.r.t. a set of equations, denoted by Th_{pfa}. Intuitively, the evaluation of join point is the evaluation of product family algebra terms in the quotient term algebra $T_{\mathcal{F}}(\mathfrak{L}(S))_{/Th_{pfa}}$. Hence, a key to match the kind pointcut is to identify the theory Th_{pfa} associated with a given base specification and a given aspect. Primarily, with regard to different types of kind pointcuts, the evaluations of product family algebra terms are corresponding to two different sets of equations. When the type of the kind pointcut is equivalent_component, a product family algebra term should be evaluated w.r.t. the actualised algebraic specification as given in Equation (6.3), while otherwise, a product family algebra term is evaluated w.r.t. the parametrised algebraic specification as given in Equation (6.1). Formally, we define $E\_tk \subseteq Th_{pfa}$ such that

$$E\_tk = \begin{cases} E_f \cup Eq(S) & \text{when } tk = \textsf{equivalent\_component} \\ E_f & \text{otherwise} \end{cases} \tag{6.12}$$

where tk denotes the type of the kind pointcut.

Moreover, if the join point is matched according to a through scope of v, extra equations need to be added into the theory Th_{pfa}. Let $\mathrm{F}(l) = context_{\mathcal{DJ}}(l) \cap context\_list\big(\{v\}, Eq(S)\big)$ record the predecessors of a label l in the hierarchical flow of the specified scope v. Consequently, we obtain the path of the hierarchical

flow from v to a matched join point p as follows:

$$hierarchy\_path(p) = \{p\} \cup \{hierarchy\_path(x) \mid x \in \mathrm{F}(p)\} \ . \tag{6.13}$$

Then the set of additional equations w.r.t. to a particular join point p, which is denoted by $E\_add(p)$, is defined as follows:

$$E\_add(p) = \{(X, l, r) \in Eq(S) \mid \exists(l \mid: l \in hierarchy\_path(p))\}. \tag{6.14}$$

Example 6.2. *We continue evaluating the join point* `full_base_functionality` *(at Line* 9 *of the* base specification*) against the kind pointcut of the aspect* case_6.1 *according to Equation (6.13).*

$$\begin{aligned}
& hierarchy_path(\texttt{full_base_functionality}) \\
= \ & \{\texttt{full_base_functionalityr}\} \\
& \cup\{hierarchy_path(x) \mid x \in F(\texttt{full_base_functionality})\} \\
= \ & \{\texttt{full_base_functionality}\} \cup hierarchy_path(\texttt{move_control}) \\
= \ & \{\texttt{full_base_functionality}, \texttt{move_control}\} \\
& \cup\{hierarchy_path(x) \mid x \in F(\texttt{move_control})\} \\
= \ & \{\texttt{full_base_functionality}, \texttt{move_control}\} \cup \{hierarchy_path(x) \mid x \in \emptyset\} \\
= \ & \{\texttt{full_base_functionality}, \texttt{move_control}\}
\end{aligned}$$

Therefore, $E\_add($`full_base_functionality`$)$ *will include equations corresponding to Line* 8 *in* base specification*. Then we would see that* `move_control` *is a subterm of* `full_base_functionality` *in Line* 9 *w.r.t.* $E\_tk \cup E\_add($`full_base_functionality`$)$*, which indicates that the kind pointcut expression is matched.*

The ⟨expression⟩ pointcut works as a guard that captures customised properties of the product family. Straightforwardly, the predicate expression() is defined as the application of the customised Boolean expression to the join point concerned. Furthermore, the expressiveness of the Boolean expression is decided by functions that can be evaluated by *Jory*.

Example 6.3. *Using Jory, we can obtain* $\#(\textit{full\_base\_functionality}) = 1$. *Then with regard to the Boolean expression of the aspect* case_6.1*, we have*

$$\big(\#(\textit{full\_base\_functionality}) \geq 1\big) \Leftrightarrow \mathsf{true}$$

According to the above discussion, the three predicates used in (6.4) are precisely given as follows:

$$\begin{cases} kind(p) & \overset{\text{def}}{\Longleftrightarrow} & (E\_tk \cup E\_add(p) \models e \preceq p) \\ expression(p) & \overset{\text{def}}{\Longleftrightarrow} & Boolean\_exp(p) \\ scope(p) & \overset{\text{def}}{\Longleftrightarrow} & matchscope(scope\_exp, p) \end{cases} . \tag{6.15}$$

where *scope_exp*, *Boolean_exp*, respectively denote the first and second components of the pointcut triple of an aspect, and e denotes a product family algebra term that is decided by the third component of the pointcut triple. According to Equation (6.15), and Examples 6.1, 6.2 and 6.3, we illustrate that `full_functionality_elevator` at Line 9 of base specification is selected as an actual join point w.r.t. the aspect case_6.1.

Advice Binding

The advice equation of an aspect in AO-PFA may include a variable jp, which needs to be instantiated before being introduced to the base specification. Intuitively, the instantiation of an advice is to replace the variable jp by the concrete term of the join point. Formally, the instantiation of the advice is represented as $a[jp/e]$, where a represents the term of the advice and e represents the term of a concrete join point. Moreover, the terms a and e are implicitly decided by different pointcut kinds, which are further explained later.

Weaving Execution

To unify the weaving process with regard to different types of pointcuts, we consider the execution of weaving process as two consequent sub-processes: introducing the advice at selected definition join points, and introducing the advice at selected reference join points.

Introducing the advice at definition join points

We firstly consider the pointcut matching for potential definition join points ($p \in \mathcal{DJ}$). As discussed in Chapter 4 and Chapter 5, a kind pointcut expression is composed by the type of kind pointcut tk and a term $k \in T_{\mathcal{F}}(V)$. Then the specified forms for matching definition join points are collected by a set, denoted by dj, as follows:

$$dj = \begin{cases} context_{\mathcal{DJ}}(k) & \text{when } tk = \mathsf{component\_creation} \\ k & \text{when } tk \in \{\mathsf{declaration}, \mathsf{creation}\} \\ \emptyset & \text{otherwise} \end{cases} \tag{6.16}$$

In other words, an element of dj corresponds to e in (6.15), and $kind(p)$ can be simplified as $p \in dj$. Moreover, we do not need to evaluate $scope(p)$ and $expression(p)$ for definition join point. Consequently, the advice is introduced by inserting a new line as below:

$$aspectID[jp/e] = Advice(jp)[jp/e].$$

Example 6.4. *To illustrate the three cases in Equation (6.16), we revisit two other examples of aspects introduced in Chapter 4 (on page 77, and 73), and denote them as* case_6.2*, and* case_6.3.

case_6.2:	
Aspect	*jp_new = (1+failure_capture) · (1+ light_reset)*
where	*jp* ∈ (*base, true,* declaration*(service)*)

case_6.3:	
Aspect	*jp_new = jp · failure_capture*
where	*jp* ∈ (*base, true,* component_creation*(full_base_functionality)*)

According to Equation (6.16), the set dj is empty for the aspect case_6.1 *(since its kind pointcut is of the type* inclusion*), which indicates that we do not need to match and introduce the advice at definition join points for this case.*

For case_6.2*, the set dj is* $\{\texttt{service}\}$*, and we have* $p \in dj$ *at Line* 5 *of the base specification, and then we can insert a new line*

$$\begin{aligned} & jp\_new[jp/\texttt{service}] = (1 + failure\_capture) \cdot (1 + light\_reset)[jp/\texttt{service}] \\ \Leftrightarrow \quad & \texttt{service\_new} = (1 + failure\_capture) \cdot (1 + light\_reset) \end{aligned}.$$

For case_6.3*, we have*

$$context_{\mathcal{DJ}}(\texttt{full\_base\_functionality}) = \{\texttt{move\_control}, \texttt{light\_display}\}.$$

Similarly, the matched definition join point for the aspect of case_6.3 *can be either* `move_control` *or* `light_display`*. Therefore, we insert two new lines as follows:*

$$(1) \quad jp\_new[jp/\texttt{move\_control}] = jp \cdot \texttt{failure\_capture}[jp/\texttt{move\_control}]$$
$$\Leftrightarrow \quad \texttt{move\_control\_new} = \texttt{move\_control} \cdot \texttt{failure\_capture}$$

.

$$(2) \quad jp\_new[jp/\texttt{light\_display}] = jp \cdot \texttt{failure\_capture}[jp/\texttt{light\_display}]$$
$$\Leftrightarrow \quad \texttt{light\_displayl\_new} = \texttt{light\_display} \cdot \texttt{failure\_capture}$$

Introducing the advice at reference join points

Before discussing how to introduce an aspect at reference join points, a set rj is defined below to collect a set of pairs that are derived from dj.

$$rj = \begin{cases} \{(k, Advice(jp)[jp/k])\} & \text{when } dj = \emptyset \\ \{(e, aspectID[jp/e]) \mid e \in dj\} & \text{otherwise} \end{cases} \tag{6.17}$$

With regard to a particular aspect, each element in rj corresponds to a pair (e, a) where e is the term of reference join points that are associated with the aspect, and a is the corresponding term to be introduced at those selected reference join points.

Example 6.5. *According to Equation (6.17), we can obtain rj for* case_6.1*,* case_6.2*,*

and case_6.3 *as follows:*

$$rj = \begin{cases} \{move\_control, move\_control \cdot failure\_capture\} & case\_6.1 \\ \{(service, service\_new)\} & case\_6.2 \\ \{(move\_control, move\_control\_new), & \\ (light\_display, light\_display\_new)\} & case\_6.3. \end{cases}.$$

At this point, we can unify the process of weaving aspects with all different types of kind pointcuts; weave a set of instanced aspects at reference join points. For each instanced aspect (i.e., $(e, a) \in rj$), we match the form of reference join points with e and substitute the matched join points by a. Unlike matching definition join points, the matching of reference join points is more complicated. Some reference join points (i.e., those in $\mathcal{E}$, see Equation (6.6)) need to be matched to the scope pointcut, while others (i.e., those in $\mathcal{C}$, see Equation (6.7)) need not. Moreover, with regard to different types of kind pointcut, the potential sets of reference join points are different. For kind pointcuts of inclusion, component, and equivalent_component, the potential reference join points are only those defined by $\mathcal{E}$, whereas the potential reference join points related to the constraint[*position_list*] are only those in $\mathcal{C}$. On the other hand, all reference join points defined in $\mathcal{RJ}$ are the potential reference join points when pointcut kinds are declaration, creation, and component_creation. Formally, the set of matched reference join points, denoted by *selected_jp*, can be defined as follows:

$$
\begin{aligned}
&\mathit{selected\_jp}\\
&\stackrel{\text{def}}{=}\\
&\bullet\quad \{p \in \mathcal{E} \mid \mathit{kind}(p) \wedge \mathit{scope}(p) \wedge \mathit{expression}(p)\},\\
&\qquad \text{when } tk \in \{\mathsf{inclusion}, \mathsf{component}, \mathsf{equivalent\_component}\}\\
&\bullet\quad \{p \in \mathcal{C} \mid \mathit{kind}(p) \wedge \mathit{expression}(p)\} \text{ , when } tk = \mathsf{constraint}[\mathit{position\_list}]\\
&\bullet\quad \{p \in \mathcal{E} \mid \mathit{kind}(p) \wedge \mathit{scope}(p) \wedge \mathit{expression}(p)\} \cup \{p \in \mathcal{C} \mid \mathit{kind}(p) \wedge \mathit{expression}(p)\},\\
&\qquad \text{when } tk \in \{\mathsf{declaration}, \mathsf{creation}, \mathsf{component\_creation}\}
\end{aligned}
\tag{6.18}
$$

To execute the matching process at reference join points, we define two auxiliary functions. A function *match_kp_sp*() is defined to evaluate $\mathit{kind}(p) \wedge \mathit{scope}(p)$ for each $p \in \mathcal{E}$, while a function *match_kp*() is defined to evaluate $\mathit{kind}(p)$ for $p \in \mathcal{C}$. Moreover, $\mathit{kind}(p)$ (see Equation (6.15)) is modified slightly to denote the extract subterms that matched the specified form. In particular, the two functions *match_kp*() and *match_kp_sp*() intend to extract the actual join points from a product family algebra term and denote them by the variable jp. Precisely, given a pair $(e, a) \in rj$ and a potential join point $p \in \mathcal{RJ}$, we define:

$$
\begin{aligned}
\mathit{match\_kp}(p, e) = \quad & p'[e'/jp] \quad && \text{when } E\_tk \models (e' = e) \wedge (p' = p),\\
\mathit{match\_kp\_sp}(p, e) = \quad & p'[e'/jp] \quad && \text{when } \big(E\_tk \cup E\_add(p)\big) \models (e' = e)\\
& && \qquad \wedge (p' = p)\big) \wedge \ \mathit{scope}(p) \ .
\end{aligned}
\tag{6.19}
$$

For some cases, how to obtain e' and p' in Equation (6.19) can be difficult, which corresponds to a word problem (see Definition 3.19) and is further discussed in the next section.

The Boolean expression of the pointcut is evaluated just before introducing the

advice at the join points. If the evaluation of the Boolean expression returns true, we substitute each variable *jp* by the corresponding advice term (i.e., the term a of the given pair $(e,\ a) \in rj$). If the evaluation returns false, we substitute the variable *jp* by the original term of the join point (i.e., the term e of the pair $(e,\ a) \in rj$). Formally, let $t(jp)$ be a product family term returned by *match_kp_sp*() or *match_kp*(). Introducing advice at reference join points can be represented by a term substitution as follows:

$$\begin{cases} t(jp)[jp/a] & \text{when } (expression(p) \Leftrightarrow \mathsf{true}) \ , \\ t(jp)[jp/e] & \text{otherwise} \ . \end{cases} \tag{6.20}$$

Example 6.6. *According to Equation (6.18), the example of* case_6.1 *only has potential join points in* $\mathcal{E}$*. As given earlier in Example 6.5, there is only the element* $(\texttt{move\_control}, \texttt{move\_control} \cdot \texttt{failure\_capture})$ *in* rj*. We consider the potential reference join point* `full_base_functionality` *at line* 9 *of* base_specification*. According to Equation (6.19), we have*

$$\big(E\_tk \cup E\_add(\texttt{full\_base\_functionality})\big) \models \big((\texttt{move\_control} = \texttt{move\ \_control}) \wedge\ (\texttt{move\_control} \cdot \texttt{light\_display} = \texttt{full\_base\_functionality})\big)$$

Moreover, as discussed through Examples 6.1–6.3, we have

$$scope(\texttt{full\_base\_functionality}) \Leftrightarrow \mathsf{true}.$$

Therefore,

$$\begin{aligned} & match\_kp\_sp(\texttt{full\_base\_functionality}, \texttt{move\_control}) \\ = \; & \texttt{move\_control} \cdot \texttt{light\_display}[\texttt{move\_control}/jp] \\ = \; & jp \cdot \texttt{light\_display} \end{aligned} .$$

According to Equation (6.20), since expression($\texttt{full\_functionality\_elevator}$) $\Leftrightarrow$ true, *the reference join point that is concerned becomes*

$$\begin{aligned} & \texttt{jp} \cdot \texttt{light\_display}[jp/\texttt{move\_control} \cdot \texttt{failure\_capture}] \\ \Leftrightarrow \; & \texttt{move\_control} \cdot \texttt{failure\_capture} \cdot \texttt{light\_display} \end{aligned} .$$

6.2 Theoretical Properties of the Weaving Process

As mentioned in the last section, the weaving process for AO-PFA is related to the word problem. Similar to the approach used in solving the word problem in general, I propose a rewriting system to solve the word problem associated with the weaving process for AO-PFA. Moreover, we analyze the proposed rewriting system concerning several theoretical properties of the weaving process.

6.2.1 Concerns Regarding the Weaving Process of AO-PFA

Let Th_{pfa} be the equational theory that is associated with a given base specification an aspects. We formalize two issues that require attentions for the weaving process.

Primarily, the word problem should be convergent to enable a decidable procedure for the weaving process. We state the above concern as follows:

Concern 1. *To implement the weaving process, we need a decidable procedure for the word problem* $Th_{pfa} \models (s = t)$ *where* $s, t \in T_{\mathcal{F}}(V)$.

We match the join points by extracting equivalent subterms from product family terms in the PFA specification. In particular, a subterm e can be extracted from a product family term p iff p can be represented as an equivalent term $\beta \cdot \mathrm{e} + \gamma$, where β and γ are product family algebra terms. Additionally, e cannot be further extracted from β and γ. Then the weaving of an aspect is processed by substituting the term of join point by the term of the advice. Obviously, terms β and γ should be unique w.r.t. the given p and e. Otherwise, substituting join points by an advice will cause ambiguous weaving results. However, it is easy to show that, if the term e and p are in arbitrary forms, the term β and γ cannot be always uniquely decided w.r.t. the equational theory E_f. For example, consider a product family p as $f_1^2 + f_1 \cdot f_2 + f_2^2$, and a join point term e as $f_1 + f_2$. With accordance to the equations (5), (6), and (8) in E_f, we possibly have two different equivalent terms of p, either $f_1^2 + f_2 \cdot (f_1 + f_2)$, or $f_1 \cdot (f_1 + f_2) + f_2^2$. Consequently, an aspect which removes a variation point $f_1 + f_2$, will cause ambiguous weaving results. Although composing such an aspect would not always cause problems, the possible weaving ambiguity impedes the reusability of the aspect. Therefore, instead of restricting the form of p, the term of the selected join points e should be constrained in a certain form to avoid ambiguous weaving results. We state the above concern as follows:

Concern 2. *Assume a term* e *is extracted from an arbitrary term* p *as a joint*

point. Then we have $p = \beta \cdot e + \gamma$ *w.r.t. to* Th_{pfa}*, where* e *cannot be further extracted from* β *and* γ*. To avoid ambiguous weaving results, the term* e *should be restricted to a form such that we can have unique* β *and* γ *up to the given theory* Th_{pfa}*.*

We note that β and γ can be the constant 0. For any term e, we assume that it cannot be extracted from 0.

6.2.2 Characteristics of the Rewriting System

There are procedures to construct a convergent term rewriting system corresponding to a given set of equations. The details of the procedure can be found in [BN98]. We only present below the rewriting system that is constructed in our context.

According to Equations (6.12), (6.14), and (6.15), the equational theory Th_{pfa} associated with the weaving process for AO-PFA is the union of two sets E_f (see Table 6.1), and E_{spec} (from those labeled family equations in $Eq(S)$). Rewrite rules derived from E_f are denoted by $R(E_f)$ and given in Table 6.2. The numbering at the beginning of each line illustrates its corresponding equations in the Table 6.1.

Table 6.2: *Rewriting rules* $R(E_f)$

r3	[L-R]	$\big(\{x\}, +(x, 0) \longrightarrow x\big)$
r4	[L-R]	$\big(\{x\}, +(x, x) \longrightarrow x\big)$
r7	[L-R]	$\big(\{x\}, \cdot(x, 1) \longrightarrow x\big)$
r8	[L-R]	$\big(\{x, y, z\}, \cdot(x, +(y, z)) \rightarrow +(\cdot(x, y), \cdot(x, z))\big)$
r9	[L-R]	$\big(\{x\}, \cdot(x, 0) \longrightarrow 0\big)$

For equations in E_{spec} we always get L-R-rules:

$$R(E_{spec}) \stackrel{\text{def}}{=} \{(X, L \longrightarrow R) \ /(X, L, R) \in E_{spec}\}. \tag{6.21}$$

The rewriting system for the weaving process in AO-PFA is $R(E_f) \cup R(E_{spec})$. Theorem 3.3 indicates that the word problem is decidable iff the rewrite system is convergent (i.e., both confluent and terminating). Therefore, we need to prove that the proposed term rewriting system is confluent and terminating with respect to Concern 1. Furthermore, to ensure the uniqueness of the weaving results, the form of the selected join point is restricted by the normal form of the proposed rewriting system with respect to Concern 2. In the remainder of this section, we formally analyze the above characteristics of the proposed term rewriting system.

Convergence of the rewriting system

According to the mathematical settings given in Subsection 6.1.1, the signature of a term rewriting system $R(E_f) \cup R(E_{spec})$ is over a finite signature $\{\{f\}, \{+, \cdot, 0, 1\} \cup \mathfrak{L}(S)\}$. To show the termination of the rewriting system, we first define a strict order over its signature as follows:

Convention 6.7. *Let $f_1, \ldots, f_n, F_1, \ldots, F_m$ be the declare and definition order specified in a PFA specification S, where $f_i (1 \leq i \leq n) \in \mathfrak{L}(S)$ and $F_i (1 \leq i \leq m) \in \mathfrak{L}(S)$, respectively denote the labels of basic features and families defined by the specification S and indexed according to their order of occurrence in S. We define a strict order over $\{\{f\}, \{+, \cdot, 0, 1\} \cup \mathfrak{L}(S)\}$ as follows:*

$$F_m > \cdots > F_1 > f_n > \cdots > f_1 > \cdot > + > 1 > 0.$$

According to the above convention, we can always obtain a strict order over the signature of the rewriting system $R(E_f) \cup R(E_{spec})$ for any syntactically correct PFA specification. Then based on Theorem 3.7, a rewriting system is terminating iff we can find a reduction order such that for any rewrite rule in the system the term at left-hand side is strictly greater than the term at the right hand side.

Theorem 6.1. *For any syntactically correct base specification S and aspect specification A, the rewriting system $R(E_f) \cup R(E_{spec})$ is terminating.*

Proof. According to Convention 6.7 and Definition 3.25, we can find a reduction order $>_{lpo}$ for the rewriting system. We then proved lemmas A.11–A.14 to show that for all rewrite rule $l \longrightarrow r$ in $R(E_f) \cup R(E_{spec})$, we have $l >_{lpo} r$.The theorem is then proved according to Theorem 3.7. All related proofs for the termination of the proposed rewriting system are given in Appendix A.2.1. □

The confluence of a general rewriting system is in general undecidable. However, the confluence of certain subsets of rewriting systems is decidable. We have proved the termination of the proposed rewriting system. According to Theorem 3.9, the rewriting system is confluent iff all its critical pairs (see Definition 3.27)are joinable.

Theorem 6.2. *For any syntactically correct specification S and aspect specification A, the rewriting system $R(E_f) \cup R(E_{spec})$ is confluent.*

Proof. Lemma A.16 is firstly proved to identify all potential critical pairs that are derived from $R(E_f) \cup R(E_{spec})$. Let $l_1 \longrightarrow r_1$, and $l_2 \longrightarrow r_2$ denote two rewrite rules from $R(E_f) \cup R(E_{spec})$. With regard to Definition 3.27, we need to rename the variable to ensure that $\mathcal{V}ar(l_1, r_1) \cap \mathcal{V}ar(l_2, r_2) = \emptyset$. In particular, variables in

the rewriting rule that corresponds to $l_1 \to r_1$ keep the same, while variables in the rewriting rule that corresponds to $l_2 \to r_2$ are renamed by adding the prime symbol. For example, x becomes x'. Then Lemmas A.17–A.25 are proved to show that all those potential pairs are either non-critical pairs, or joinable critical pairs. Finally, the theorem is proved according to Theorem 3.9. All proofs related to this theorem are given in Appendix A.2.2. □

The axioms 1, 2, 5, and 6 in Table 6.1 are associated with associativity and commutativity of the function symbols + and $\cdot$. The associativity and commutativity appearing in our equational theory, called AC-theory, are quite common properties for binary operations. We cannot introduce rewriting rules for the commutativity equation as the other equations in E_f, since it will lead to a nonterminating rewriting system. Consequently, rewriting rules for AC-theory will be nonterminating. On the other hand, equation unification can be used for term rewriting, which considers semantic properties of function symbols. Therefore, instead of introducing rewriting rules for the AC-theory, we use AC-unification for the corresponding term rewriting with respect to the equations 1, 2, 5, and 6 in our equational theory. Algorithms of AC-Unification can be found in [BN98], and for many rewriting tools, we can use build-in AC-Unification modulo to implement the AC Unification directly.

Definition 6.8 ($\lfloor_{AC}$)**.** *Given a set of rewrite rules R on terms in $T_{\mathcal{F}}(V)$, we call a term an AC normal term iff the term and any equivalent term of it (up to the AC theory of + and $\cdot$) cannot be further rewritten by any rule in R. For any term β, we denote its AC normal form as $\beta\lfloor_{AC}$.*

Theorem 6.3. *Let Th_{pfa} be the equational theory obtained according to the given base and aspect specifications. Then we have*

$$(Th_{pfa} \models (s = t)) \iff (R(E_f) \cup R(E_{spec}) \vdash (s \downarrow_{AC} \overset{AC}{=} t \downarrow_{AC})),$$

where $\overset{AC}{=}$ denotes the equivalent of two terms up to the AC-theory of $+$ and $\cdot$.

Proof.

$(Th_{pfa} \models (s = t))$

$\iff$ $\langle$ Definition of Th_{pfa} $\rangle$

$(\{e3, e4, e7, e8, e9\} \cup E_{spec} \cup \{e1, e2, e5, e6\}) \models (s = t)$

$\iff$ $\langle$ Lemma 3.1 $\rangle$

$(\{e3, e4, e7, e8, e9\} \cup E_{spec}) \models (\{e1, e2, e5, e6\} \implies (s = t))$

$\iff$ $\langle$ Definition 3.22 & Definition 6.8 $\rangle$

$(R(E_f) \cup R(E_{spec})) \vdash (s \downarrow_{AC} \overset{\text{AC}}{=} t \downarrow_{AC})$

□

Corollary 6.4. *Given a syntactically correct PFA base specification and an aspect specification, we can have a decidable procedure for the word problem associated with the weaving process of AO-PFA.*

Proof. Based on Theorems 6.1 and 6.2, we show that the term rewriting system $R(E_f) \cup R(E_{spec})$ for a syntactically correct PFA base specification and an aspect is convergent (i.e., terminating and confluent). According to Theorem 6.3, it is

obvious that we have a decidable procedure for the word problem with regard to Th_{pfa}, where Th_{pfa} is decided by the given PFA specifications and aspects. □

The restriction on the selected join points

As we have shown earlier, the weaving results can be ambiguous if the form of join point is arbitrary. One solution to avoid ambiguous results is to extract join points from the unique [2] normal form w.r.t. the term rewriting system $R(E_f) \cup R(Eq)$. In other words, the form of join point should be restricted by the form of the normal form. Let s be any non-ground term that specified in the base specification (i.e., $s \in T_{\mathcal{F}}(\mathfrak{L}(S))$). The AC normal form of β associated with the rewriting system $R(E_f) \cup R(E_{spec})$ can be expressed as $\sum_{i=1}^{m} s_i$ such that all s_i are not equivalent (up to AC theory of $\cdot$). Furthermore, each $s_i (1 \leq i \leq m)$ is either 1, or $\prod x_{ij}$ where each $x_{ij} \in \mathfrak{L}(S)$. This result is formally stated by Lemma A.27 in Appendix A.2.3. The detailed proof is given on page 179. Basically, we use a structure inductive proof over the term s. Therefore, with the proposed rewriting system, the AC normal form of any non-ground term in the base specification S is a sum of products of labels from $\mathfrak{L}(S)$. Intuitively, a term in the form of a product is always a subterm of $s \downarrow_{AC}$ if the term can be extracted form s. In other terms, the AC normal form of the term e w.r.t. Concern 2 should be a product of labels from $\mathfrak{L}(S)$.

Theorem 6.5. *Let e and p be two given product family terms. Assume that w.r.t. the set of equations Th_{pfa}, $p = \beta_1 \cdot e + \gamma_1 = \beta_2 \cdot e + \gamma_2$, where e cannot be further extracted from β_1, β_2, γ_1 and γ_2. Then if $e \downarrow_{AC}$ is a product of labels, we have $\beta_1 = \beta_2 \wedge \gamma_1 = \gamma_2$ w.r.t. to the set of equations Th_{pfa}.*

[2] Uniqueness here means that two normal form are equal up to the AC-theory of $\cdot$ and $+$.

Proof. The proof of this theorem is based on several lemmas given in Appendix A.2.3. Lemma A.28 states that e cannot be extracted from any term s in the base specification iff e is not dividable by any product of $s \lfloor_{AC}$. Lemma A.29 gives the formal representation of $(\beta \cdot e + \gamma) \lfloor_{AC}$, while Lemma A.30 indicates the exact relationships between $s \lfloor_{AC}$ and $t \lfloor_{AC}$ iff $s = t$ w.r.t. Th_{pfa}. All proofs related to this theorem are given on pages 183–186. □

Theorem 6.5 shows the uniqueness of β and γ w.r.t. the proposed restriction on the term of selected join points. According to Concern 2 (given on Page 130), it is straightforward to see that we can obtain unambiguous weaving results by using the proposed rewriting system. Precisely, terms e, p correspond to e, p in Equation (6.19), while terms $e \lfloor_{AC}$, $p \lfloor_{AC}$ correspond to e', p' in Equation (6.19).

Corollary 6.6. *Let S be a syntactically correct base specification and A an aspect. Using the decidable procedure given in Theorem 6.4 for the weaving process, the weaving result is unambiguous if the term of join points selected by A is a product of labels from $\mathfrak{L}(S)$.*

6.3 Conclusion

In this chapter, I discussed several issues related to the automation of the weaving process for AO-PFA. I formally discussed the general models of PFA specifications and their aspects. Then I articulated the general behaviour of the weaver for AO-PFA. I also briefly discussed some theoretical properties related to the weaving process. A word problem is induced in the weaving process and tackled by constructing a rewriting system. The rewriting system then is proved to be

convergent, and the form of join points is restricted to avoid ambiguous weaving results.

Moreover, the formalisms of base specifications and aspects are validated by using the algebraic specification language CASL [BM04] as given in Appendix B.1. The proposed rewriting system is validated with a term rewriting tool called Maude [CDE+11] as given in Appendix B.2.

Chapter 7

Conclusion and Future Work

In this thesis, I present a comprehensive approach to extend product family algebra with the abilities to specify, verify, and weave aspects. The idea of adopting the aspect-oriented paradigm at the feature-modeling level aids in handling the difficulties that arise from crosscutting concerns and unanticipated changes in large-scale feature models. On one hand, the crosscutting concerns are rigorously specified by aspects and the composition of crosscutting concerns is guaranteed to be sound according to the verification and weaving techniques for aspects. On the other hand, an original feature model can be considered as a base specification, while any unanticipated changes to the feature model can be specified as aspects. Meanwhile, the formal verification and weaving techniques ensure that the changes are correctly and properly propagated to the original feature model.

7.1 Highlight of the Contributions

In this section, I highlight the contributions related to the proposed aspect-oriented approach for feature-modeling. I presented a specification language AO-PFA, which extends the language of PFA specifications by including aspects. In AO-PFA, an aspect consists of an advice equation and a pointcut triple. The proposed syntax for aspects allows us to articulate aspects in a flexible yet systematic way. I also gave a classification of aspects based on their syntax. The classification of aspects indicates the expected modifications caused by aspects on base specifications. Furthermore, I proposed a formal verification technique to check the compatibility of aspects with their base specification in AO-PFA. I extracted the validity criteria of PFA specification from three facets: definition-validity, reference-validity, and dependency-validity. By analyzing the impact of aspects on the three validity criteria, I formalized the detection technique for invalid aspects w.r.t. their base specifications. The proposed verification technique ensures that the validity of a base specification is not spoiled by composing aspects. Finally, I discussed how to perform the weaving process in AO-PFA. By describing PFA specifications as parametrized algebraic specifications, I formalized the behaviours of the weaver for AO-PFA. I also introduced a term rewriting system to solve the word problem associated with the weaving process. I proved the convergence of the induced rewriting system, and showed that the selected join points should be in a certain form to avoid ambiguous weaving results. With the provided term rewriting system and the restriction on join points, we can obtain a deterministic procedure to automate the weaving process in AO-PFA.

In [ACLF10], we find that inserting new requirements into a feature model or

merging several feature models are the two essential operations for evolving feature models. With the proposed aspect-oriented approach for feature-modeling, merging feature models can be handled by using view reconciliation as presented in [HKM08, HKM11a]. Inserting new requirements into a feature model can be handled with the aspect-oriented paradigm. From another perspective, my study explored the relationship between original feature models and modified feature models by specifying aspects modularly and classifying aspects according to their effects on a base specification. Another focus of my study is formalization. An algebraic foundation for the feature composition is proposed in [ALMK10], which algebraically captures the basic ideas of features and the feature composition. Whereas their approach is more related to programming languages at the implementation stage, the approach proposed in this thesis resides in the early analysis and design stages of product family engineering. My study fills the gaps in the formalization of the product family engineering at different stages.

Furthermore, by properly handling the crosscutting concerns and anticipated changes in feature models, the work presented in this thesis might contribute to deal with different issues in feature-oriented software development. At the domain analysis stage of feature-oriented software development, the proposed aspect-oriented approach for feature-modeling can be used to represent *large-scale feature models* by separately specifying base specifications and aspects. Besides, the proposed approach also enables one to capture changes on feature models by specifying aspects separately from the original feature models, which can benefit the *evolution of feature models*. Moreover, the earliest research about handling *feature composition and interaction* problems is conventionally investigated at the domain

design/specification stage of feature-oriented software development when reifying each feature. With the proposed approach, the potential feature composition and interaction problems can be discussed starting from a higher level by analyzing the dependencies of aspects and the weaving process. The detection of invalid aspectual composition and restriction on weaving of aspects might imply useful information for deriving a suitable strategy to compose features at later stages.

7.2 Further Work

In the following, I list some of the possible future extensions to this work from different directions. The aim is to extend the proposed approach to different stages of product family engineering. I also discuss the application of the proposed approach, and the tools to automate it.

7.2.1 Theory: Models and Techniques

A number of future research can be conducted as a result of the proposed feature-modeling technique.

(i) The proposed specification language AO-PFA might be enriched to have a more comprehensive expressiveness for specifying feature models. For example, we can have a construct "include" to improve the modularity and extensibility of specifications. The key of the language enrichment is to keep a balance between the modeling power and the analyzability of the language.

(ii) The translation between product family algebra and other diagram-based

feature-modeling techniques has been discussed in the literature. By extending product family algebra with the aspect-oriented paradigm, it might also be interesting to analyze how the aspect-oriented paradigm is related to other digram-based feature-modeling techniques. This work will involve an investigation into other digram-based feature-modeling techniques with the ability of composing feature models.

(iii) A key ultimate goal of the feature modeling is the automatic code generation of products from the specification of the base product family, the specification of the aspects, and the specification of each of the basic features. In [HKM11b], the features of a product family were given as requirements scenarios formalized as pairs of relational specifications of a proposed system and its environment. We can use the work presented in [HKM11b] as the basis for our future work to introduce finer granularity aspects at the state level and then generate the code of a product.

7.2.2 Application

Different applications can be investigated with the proposed technique.

(i) It is necessary to apply the proposed approach to real-world scenarios for developing product families. In this thesis, only small case studies are used to illustrate the flexibility of the proposed specification language. On the other hand, the language is designed to capture product families with huge number of features. By using the language and its verification and weaving techniques on industrial-sized feature models, the empirical evaluation of the proposed approach gives us a better idea on its real strengths and weaknesses.

(ii) Although my study focuses on the domain analysis of product family engineering, there is theoretically no obstacle to apply the proposed feature-modeling technique to other domains. For example, security is a common issue recognized as a crosscutting concern, and making unanticipated changes related to security requirements is also unavoidable [LRCe06]. Various security mechanisms (e.g., [HLN04, RFLG04, SH04, LRCe06]) have been proposed based on the aspect-oriented paradigm. The proposed aspect-oriented specification language can be applied to those security mechanisms in terms of feature models. With the help of the verification and weaving techniques, we can recognize some interrelations between security concerns with other concerns, and identify the possible conflicts among them.

7.2.3 Tool/Automation

The existing *Jory* tool can be extended by implementing the proposed approach. The envisioned extension would use the kernel of *Jory* for the automatic analysis of large feature models. We only need to perform a lightweight extension to the existing notation of *Jory* to support the aspect specifications. In addition, as *Jory* is designed to be extensible, two modules can be added regarding the verification of aspectual composition and the weaving of aspects.

7.3 Closing Remarks

The systematic development with the aspect-oriented paradigm is an on-going area with many opening issues. It is still unseen how well the aspect-oriented software development can work in the context of feature-oriented software development.

How to map the aspect-oriented paradigm at feature-modeling to the implementation level is still a struggle for both the communities of product family engineering and aspect-oriented software development. Therefore, it remains a complex task towards the systematic development and automatic derivation of products using the aspect-oriented paradigm. However, by starting from the feature-modeling level, the proposed approach for aspect-oriented feature-modeling at least helps to improve, from an early stage, the traceability of crosscutting concerns and unanticipated changes.

Appendix A

Lemmas, Theorems, and Corollaries

A.1 Proofs of the Results of Chapter 5

Theorem A.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.

1. u and v are *mutually defined*

   $\iff$ $\langle$ Definition of mutually defined $\rangle$

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

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

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

$\Longleftrightarrow$ $\langle$ Dummy renaming $\rangle$

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

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

There is a cycle including u and v in G_{S}

2. u is self-defined

$\Longleftrightarrow$ $\langle$ Definition of mutually defined & $u = v$ & $m = n = 1$ $\rangle$

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

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

(u,u)–path $\in E$

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

There is a loop through u in G_{S}

□

Theorem A.2 (Kind Pointcut 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 A is constraint$[list] \implies (D_{\mathrm{A}} = \mathrm{E_add}_{\mathrm{A}} = \mathrm{E_del}_{\mathrm{A}} = \emptyset)$. We then prove $D_{\mathrm{A}} = \mathrm{E\_add}_{\mathrm{A}} = \mathrm{E\_del}_{\mathrm{A}} = \emptyset \implies A$ is dependency-valid w.r.t. S.

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

$\Longleftrightarrow$ $\langle$ Definition 5.10 $\rangle$

$\forall(u,v \mid u,v \in D_{\mathrm{S}'} : \neg \mathit{mutdef}(u,v))$

$\Longleftrightarrow$ $\langle$ Definition of *mutdef* (See page 93) $\rangle$

$\forall(u,v \mid u,v \in (D_{\mathrm{S}} \cup D_{\mathrm{A}}) : \neg \exists(m,n \mid m,n \geq 1 : (u,v)\text{–path} \in (\mathrm{E\_add}_{\mathrm{A}}$

$$\cup E_S - \text{E\_del}_\text{A})^m \wedge (u,v)\text{–path} \in (\text{E\_add}_\text{A} \cup E_S - \text{E\_del}_\text{A})^n \,)\,)$$

$\iff$ $\langle$ Assumption: $D_\text{A} = \text{E\_add}_\text{A} = \text{E\_del}_\text{A} = \emptyset$ $\rangle$

$$\forall(u,v \mid u,v \in D_\text{S} : \neg\, \exists(m,n \mid m,n \geq 1 : (u,v)\text{–path} \in (E_S)^m \wedge (v,u)\text{–path} \in (E_S)^n \,)\,)$$

$\iff$ $\langle$ Definition of *mutdef* $\rangle$

$$\forall(u,v \mid u,v \in D_\text{S} : \neg \mathit{mutdef}(u,v)\,)$$

$\iff$ $\langle$ S is dependency-valid (Definition 5.6) $\rangle$

true

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

Theorem A.3 (Non-cycle Condition)**.** *Let S be a valid PFA specification and A be an aspect that does not satisfy the kind pointcut condition (Theorem 5.1). Construct the label dependency digraph G_S according to Construction 5.5 and denote or create the vertex k in G_S according to Construction 5.11. Then A is dependency-valid w.r.t. S if $\forall(x \mid x \in D_S \cap R_A : \mathit{Walk}(k,x) = \emptyset\,)$.*

Proof. Let A_def be a set of new labels assigned to the left-hand sides of labelled family equations. Let B_def be the set of all labels at the left-hand sides of all labelled family equations where join points are present at their right-hand sides. The edges in E_add_A generate a path from each vertex $u \in R_\text{A}$ to each vertex $v \in \text{B}_\text{def}$ in the amended label dependency digraph. Moreover, each edge in E_del_A is always ended with a vertex $v \in \text{B}_\text{def}$ in the amended label dependency digraph. Due to the ordering properties of the dependency digraph, edges in E_del_A would

not be within a path that starts from a vertex in $\mathrm{B_{def}}$ in the original dependency digraph. Therefore, A is dependency-invalid w.r.t. S iff

$$\exists(u, v \mid u, v \in D_{\mathrm{S}} : u \in \mathrm{B_{def}} \wedge v \in R_{\mathrm{A}} \wedge \exists(m \mid m \geq 1 : (u, v)\text{–path} \in (E_S)^m))$$
$$\vee \exists(u \mid u \in D_{\mathrm{S}} : u \in \mathrm{B_{def}} \wedge u \in R_{\mathrm{A}}) \tag{A.1}$$

In Exp. A.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.

Exp. A.1

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

$$\exists(u \mid u \in D_{\mathrm{S}} \wedge u \in R_{\mathrm{A}} : u \in \mathrm{B_{def}})$$
$$\vee \exists\big(v \mid v \in D_{\mathrm{S}} \wedge v \in R_{\mathrm{S}} : \exists(u \mid u \in \mathrm{B_{def}} : \exists(m \mid m \geq 1 :$$
$$(u, v)\text{–path} \in (E_S)^m))\big)$$

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

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B_{def}} \vee \exists(u \mid u \in \mathrm{B_{def}} : \exists(m \mid m \geq 1 :$$
$$(u, v)\text{–path} \in (E_S)^m))\big)$$

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

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \exists(n \mid n \geq 1 : (k, v)\text{–path} \in (E_S)^n)$$
$$\vee \exists(u \mid : \exists(n \mid n \geq 1 : (k, u)\text{–path} \in (E_S)^n)$$
$$\wedge \exists(m \mid m \geq 1 : (u, v)\text{–path} \in (E_S)^m))\big)$$

$$\Longleftrightarrow \quad \langle \text{ Path concatenation } \& \text{ Dummy renaming } \& \text{ Distributivity of } \wedge \text{ over } \exists \ \rangle$$

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \vee \big(\exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m) \wedge \exists(u \mid u \in D_{\mathrm{S}} : \exists(n \mid n \geq 1 : (k,u)\text{–path} \in (E_S)^n))\big)\big)$$

$$\Longrightarrow \quad \langle \text{ Weakening } \rangle$$

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n) \vee \exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m)\big)$$

$$\Longleftrightarrow \quad \langle \text{ Dummy renaming } \& \text{ Range split } \& \text{ Idempotency of } \vee \rangle$$

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \exists(n \mid n \geq 1 : (k,v)\text{–path} \in (E_S)^n)\big)$$

$$\Longleftrightarrow \quad \langle \text{ Definition of } \mathit{Walk}(u,v) \ \rangle$$

$$\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \mathit{Walk}(k,v) \neq \emptyset)$$

Therefore,

$$A \text{ is dependency-invalid w.r.t. } S \implies \exists(x \mid x \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \mathit{Walk}(k,x) \neq \emptyset)$$

If we take its contrapositive form, we have

$$\forall(x \mid x \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \mathit{Walk}(k,x) = \emptyset) \implies A \text{ is dependency-valid w.r.t. } S$$

□

From the above proof, we can obtain an proposition as follows:

$$
\begin{aligned}
& A \text{ is dependency-invalid w.r.t. } S \\
\Longleftrightarrow\ & \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B}_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{B}_{\mathrm{def}} : \exists(m \mid m \geq 1 : \\
& (u,v)\text{–path} \in (E_S)^m)\,)\big) \qquad \text{(A.2)}
\end{aligned}
$$

Lemma A.4. *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 type of the scope pointcut is *base*, join points are where k is present. Therefore, $\mathrm{B}_{\mathrm{def}} = N^+(k)$.

$$
\begin{aligned}
& A \text{ is dependency-invalid w.r.t. } S \\
\Longleftrightarrow\ & \langle\ \text{A.2} \quad \& \quad \mathrm{B}_{\mathrm{def}} = N^+(k)\ \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in N^+(k) \vee \exists(u \mid u \in N^+(k) : \exists(m \mid m \geq 1 : \\
& \qquad (u,v)\text{–path} \in (E_S)^m)\,)\big) \\
\Longleftrightarrow\ & \langle\ x \in N^+(k) \iff (k,x)\text{–path} \in E_S\ \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : (k,v)\text{–path} \in E_S \vee \exists(u \mid u \in D_{\mathrm{S}} : (k,u)\text{–path} \in E_S \\
& \qquad \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\,)\big) \\
\Longleftrightarrow\ & \langle\ \text{Path concatenation} \quad \& \quad \text{Distributivity of } \wedge \text{ over } \exists\ \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : (k,v)\text{–path} \in E_S \vee \big(\exists(u \mid u \in D_{\mathrm{S}} : (k,u)\text{–path} \in E_S\,) \\
& \qquad \wedge \exists(m \mid m \geq 2 : (k,v)\text{–path} \in (E_S)^m)\big)\big)
\end{aligned}
$$

$\Longleftrightarrow$ $\langle$ Theorem 5.2 does not hold $\Longrightarrow$ $\exists(u \mid u \in D_{\mathrm{S}} : (k,u)$–path $\in E_S)$ & Identity of $\wedge$ $\rangle$

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

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

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

$\Longleftrightarrow$ $\langle$ Definition of $Walk(u,v)$ & Precondition: Theorem 5.2 does not hold $\rangle$

true

□

Lemma A.5. *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 type of scope pointcut is protect(*base*), the set $\mathrm{B}_{\mathrm{def}}$ is empty.

A is dependency-invalid

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B}_{\mathrm{def}} = \emptyset$ $\rangle$

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \mathsf{false} \vee \exists(u \mid \mathsf{false} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))\big)$

$\Longleftrightarrow$ $\langle$ Empty range & $\exists$-False Body: $\exists(x \mid R : \mathsf{false}) \Longleftrightarrow \mathsf{false}$ $\rangle$

false

□

Lemma A.6. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 5.5*

*and denote or create the vertices k and s in G_S according Construction 5.11. 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 type of the scope pointcut is within, join points are bound to a labelled product equations whose label is s. Therefore, we have $\mathrm{B_{def}} = s$. Besides, there should be a path form k to s.

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B_{def}} = s$ & There is a path from k to s $\rangle$

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} \ : v = s \vee \ \exists(u \mid u = s \ : \ \exists(m \mid m \geq 1 \ : \ (u,v)\text{–path} \in (E_S)^m\,)\,)\big) \wedge \ \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)$

$\Longleftrightarrow$ $\langle$ Distributivity of $\wedge$ over $\exists$ & One-point rule $\rangle$

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} \ : (v = s \vee \ \exists(m \mid m \geq 1 \ : \ (s,v)\text{–path} \in (E_S)^m\,)) \wedge \exists(n \mid n \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^n\,)\big)$

$\Longleftrightarrow$ $\langle$ Definition of $\mathit{Walk}(u,v)$ & S is dependency-valid $\Longleftrightarrow$ $\mathit{Walk}(x,x) = \emptyset$ $\rangle$

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} \ : (s = v \vee (\,\mathit{Walk}(s,v) \neq \emptyset \ \wedge \ s \neq v)) \wedge (\,\mathit{Walk}(k,s) \neq \emptyset\,) \wedge k \neq s)\big)$

$\Longleftrightarrow$ $\langle$ Absorbing: $p \vee (q \wedge \neg p) \ \Longleftrightarrow \ (p \vee q)$ $\rangle$

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} \ : \ (s = v \vee \ \mathit{Walk}(s,v) \neq \emptyset) \wedge \ \mathit{Walk}(k,v) \neq \emptyset \wedge k \neq s\,\big)$

$\Longleftrightarrow$ $\langle$ Path concatenation $\rangle$

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} \ : \ s \in \mathit{Walk}(k,v) \wedge k \neq s\,\big)$

□

Lemma A.7. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 5.5 and denote or create the vertex k in G_S according Construction 5.11. When the scope of the pointcut is "*protect*(*within*)", A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A \ : \ s \notin Walk(k, v) \vee s = k\,)$.*

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

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B_{def}}$ properties & Lemma A.4 $\rangle$

$\big(\ \exists(k \mid k \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^k\,) \wedge \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} \ : \ (\ \exists(n \mid n \geq 1 \ :$

$(k,v)\text{–path} \in (E_S)^n\,) \wedge v \neq s)$

$\vee\ \exists(u \mid \exists(n \mid n \geq 1 \ : \ (k,u)\text{–path} \in (E_S)^n\,) \wedge u \neq s \ : \ \exists(m \mid m \geq 1 \ :$

$(u,v)\text{–path} \in (E_S)^m\,)\,)\big)$

$\vee(\neg\ \exists(k \mid k \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^k\,) \ \wedge\ \mathsf{true})$

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

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} \ : \ (\ \exists(n \mid n \geq 1 \ : \ (k,v)\text{–path} \in (E_S)^n\,) \wedge v \neq s) \vee\ \exists(u \mid$

$\exists(n \mid n \geq 1 \ : \ (k,u)\text{–path} \in (E_S)^n\,) \wedge u \neq s \ : \ \exists(m \mid m \geq 1 \ :$

$(u,v)\text{–path} \in (E_S)^m\,)\,)\big)$

$\vee\neg\ \exists(k \mid k \geq 1 \ : \ (k,s)\text{–path} \in (E_S)^k\,)$

$$\begin{array}{ll}
\Longleftrightarrow & \langle \text{ Trading rule } \ \& \ \text{ Definition of } \mathit{Walk}(u,v) \ \& \ \text{ Path concatenation } \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : (\mathit{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee \exists(u \mid u \neq s : \\
& \quad u \in \mathit{Walk}(k,v) \wedge u \neq k \wedge u \neq v)\big) \vee \mathit{Walk}(k,s) = \emptyset \\
\Longleftrightarrow & \langle \text{ Trading rule } \ \& \ \text{ Generalized De Morgan } \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : (\mathit{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee \neg \forall(u \mid u = s : \\
& \quad u \in \mathit{Walk}(k,v) \wedge u \neq k \wedge u \neq v)\big) \vee \mathit{Walk}(k,s) = \emptyset \\
\Longleftrightarrow & \langle \text{ One-point rule } \ \& \ \text{ De Morgan } \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : (\mathit{Walk}(k,v) \neq \emptyset \wedge v \neq s) \vee (s \notin \mathit{Walk}(k,v) \vee s = k \vee \\
& s = v)\big) \vee \mathit{Walk}(k,s) = \emptyset \\
\Longleftrightarrow & \langle \text{ Distributivity of } \vee \text{ over } \wedge \ \& \ \text{ Excluded Middle } \ \& \ \text{ Absorbing: } p \wedge (p \vee q) \iff p \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin \mathit{Walk}(k,v) \vee s = k\big)\big) \vee \mathit{Walk}(k,s) = \emptyset \\
\Longleftrightarrow & \langle \ \mathit{Walk}(k,s) = \emptyset \implies \exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin \mathit{Walk}(k,v)) \ \& \\
& \quad \text{Distributivity of } \vee \text{ over } \exists \ \& \ (p \implies q) \implies ((p \vee q) \equiv q) \ \rangle \\
& \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin \mathit{Walk}(k,v) \vee s = k\big)\big)
\end{array}$$

□

Lemma A.8. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the dependency digraph G_S according Construction 5.5 and denote or create the vertices k and s in G_S according Construction 5.11. When the scope of the pointcut is "*through*", 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 type of the scope pointcut is through, join points are bound to a labelled family equation where s is present at their right-hand sides. Therefore, $\mathrm{B}_{\mathrm{def}} = N^+(s)$. Besides, there should be a path from k to s.

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B}_{\mathrm{def}} = N^+(s)$ & There is a path from k to s $\rangle$

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in N^+(s) \vee \big(\exists(u \mid u \in N^+(s) : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\big)\big) \wedge \exists(n \mid n \geq 1 : (k,s)\text{–path} \in (E_S)^n)$

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

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : ((s,v)\text{–path} \in E_S \vee \exists(u \mid: (s,u)\text{–path} \in E_S \wedge \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m))) \wedge \exists(n \mid n \geq 1 : (k,s)\text{–path} \in (E_S)^n)\big)$

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

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

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

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

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

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \in Walk(k,v) \wedge s \neq v \wedge k \neq s\big)$

□

Lemma A.9. *Let S be a valid PFA specification and A be a potentially dependency-invalid aspect. Construct the label dependency digraph G_S according Construction 5.5 and denote or create the vertices k and s in G_S according Construction 5.11. When the scope of the pointcut is "*protect*(*through*)", A is dependency-invalid w.r.t. S iff $\exists(v \mid v \in D_S \cap R_A : s \notin Walk(k, v) \vee s = k \vee s = v)$.*

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

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B_{def}}$ properties & Lemma A.4 $\rangle$

$\big(\exists(k \mid k \geq 1 : (k, s)\text{–path} \in (E_S)^k)$

$\wedge \exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} : \exists(n \mid n \geq 1 : (k, v)\text{–path} \in (E_S)^n) \wedge v \notin N^+(s)$

$\vee \exists(u \mid \exists(n \mid n \geq 1 : (k, u)\text{–path} \in (E_S)^n) \wedge u \notin N^+(s) : \exists(m \mid$

$m \geq 1 : (u, v)\text{–path} \in (E_S)^m)\big)\big)$

$\vee(\mathsf{true} \wedge \neg\, \exists(k \mid k \geq 1 : (k, s)\text{–path} \in (E_S)^k))$

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

$\exists\big(v \mid v \in D_\mathrm{S} \cap R_\mathrm{A} : \exists(n \mid n \geq 1 : (k, v)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 :$

$(k, s)\text{–path} \in (E_S)^k) \wedge (s, v)\text{–path} \notin E_S) \vee \exists(u \mid \exists(n \mid n \geq 1 :$

$(k, u)\text{–path} \in (E_S)^n) \wedge \exists(k \mid k \geq 1 : (k, s)\text{–path} \in (E_S)^k)$

$\wedge (s, u)\text{–path} \notin E_S : \exists(m \mid m \geq 1 : (u, v)\text{–path} \in (E_S)^m))\big)$

$$\vee\neg\ \exists(k\ \mid\ k \geq 1\ :\ (k,s)\text{–path} \in (E_S)^k\,)$$

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

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

$$\vee\neg\ \exists(k\ \mid\ k \geq 1\ :\ (k,s)\text{–path} \in (E_S)^k\,)$$

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

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

$$\vee\neg\ \exists(k\ \mid\ k \geq 1\ :\ (k,s)\text{–path} \in (E_S)^k\,)$$

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

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

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

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

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

$$\exists\big(v\ \mid\ v \in D_{\mathrm{S}} \cap R_{\mathrm{A}}\ :\ s \notin Walk(k,v) \vee s = k \vee s = v\big) \vee Walk(k,s) = \emptyset)$$

$$\iff \quad \langle\ Walk(k,s) = \emptyset \implies \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin Walk(k,v)\big)\ \rangle$$

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin Walk(k,v) \vee s = k \vee s = v\big)$$

□

Theorem A.10 (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 Theorem 5.2. Vertices k and s are denoted or created in G_S according to A as prescribed in Construction 5.11. 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*

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

Proof.

(1) $ts = base$

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

$\iff$ ⟨ Lemma A.4 & a is a vertex associated to A that invalidates Theorem 5.2 ⟩

true

(2) $ts =$ within

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

$\Longleftrightarrow$ $\langle$ Lemma A.6 $\rangle$

$\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \in \mathit{Walk}(k, v) \wedge s \neq k)$.

$\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Theorem 5.2 $\rangle$

$s \in \mathit{Walk}(k, a) \wedge s \neq k$

(3) $ts =$ through

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

$\Longleftrightarrow$ $\langle$ Lemma A.8 $\rangle$

$\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \in \mathit{Walk}(k, v) \wedge s \neq k \wedge s \neq v)$

$\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Theorem 5.2 $\rangle$

$s \in \mathit{Walk}(k, a) \wedge s \neq k \wedge s \neq a$

(4) ts=protect(ts')

- ts'=*base*

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

  $\Longleftrightarrow$ $\langle$ Lemma A.5 $\rangle$

  false

  $\Longleftrightarrow$ $\langle$ Proof item (1) $\rangle$

  $\neg$Dep_invalid(*base*)

- ts'=within

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

  $\Longleftrightarrow$ $\langle$ Lemma A.7 $\rangle$

  $\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin \mathit{Walk}(k, v) \vee s = k)$

  $\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Theorem 5.2 $\rangle$

  $s \notin \mathit{Walk}(k, a) \vee s = k$

  $\Longleftrightarrow$ $\langle$ Proof item (2) $\rangle$

  ¬Dep_invalid(within)

- ts'=through

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

  $\Longleftrightarrow$ $\langle$ Lemma A.9 $\rangle$

  $\exists(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : s \notin \mathit{Walk}(k, v) \vee s = k \vee s = v)$

  $\Longleftarrow$ $\langle$ Witness: a is a vertex associated to A that invalidates Theorem 5.2 $\rangle$

  $s \notin \mathit{Walk}(k, a) \vee s = k \vee s = a$

  $\Longleftrightarrow$ $\langle$ Proof item (3) $\rangle$

  ¬Dep_invalid(through)

(5) $ts = (ts_1{:}ts_2)$

Let the set of join points selected by ts_1 be $\mathrm{B}^1_{\mathrm{def}}$ and the set of join points selected by ts_2 be $\mathrm{B}^2_{\mathrm{def}}$, then the set $\mathrm{B}_{\mathrm{def}} = \mathrm{B}^1_{\mathrm{def}} \cup \mathrm{B}^2_{\mathrm{def}}$.

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B_{def}} = \mathrm{B^1_{def}} \cup \mathrm{B^2_{def}}$ $\rangle$

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B^1_{def}} \cup \mathrm{B^2_{def}} \vee \exists(u \mid u \in \mathrm{B^1_{def}} \cup \mathrm{B^2_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\,)\big)$

$\Longleftrightarrow$ $\langle$ Range split for idempotent operator $\exists$ $\rangle$

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B^1_{def}} \cup \mathrm{B^2_{def}} \vee \exists(u \mid u \in \mathrm{B^1_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path}) \in (E_S)^m) \vee \exists(u \mid u \in \mathrm{B^2_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\,)\big)$

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

$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B^1_{def}} \vee \exists(u \mid u \in \mathrm{B^1_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\,)\big) \vee \exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B^2_{def}} \vee \exists(u \mid u \in \mathrm{B^2_{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\,)\big)$

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

$\mathrm{Dep\_invalid}(ts_1) \vee \mathrm{Dep\_invalid}(ts_2)$

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

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

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

$\Longleftrightarrow$ $\langle$ A.2 & $\mathrm{B_{def}} = \mathrm{B^1_{def}} \cap \mathrm{B^2_{def}}$ $\rangle$

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : v \in \mathrm{B}^1_{\mathrm{def}} \cap \mathrm{B}^2_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{B}^1_{\mathrm{def}} \cap \mathrm{B}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\big)\big)$$

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

$$\exists\big(v \mid v \in D_{\mathrm{S}} \cap R_{\mathrm{A}} : \big(v \in \mathrm{B}^1_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{B}^1_{\mathrm{def}} \cap \mathrm{B}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\big) \wedge \big(v \in \mathrm{B}^2_{\mathrm{def}} \vee \exists(u \mid u \in \mathrm{B}^1_{\mathrm{def}} \cap \mathrm{B}^2_{\mathrm{def}} : \exists(m \mid m \geq 1 : (u,v)\text{–path} \in (E_S)^m)\big)\big)\big)$$

$\impliedby$ $\langle$ Witness: a is a vertex associated to A that invalidates Theorem 5.2 $\rangle$

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

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

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

$\iff$ $\langle$ A.2 & Proof item (1), (2), (3), and (4) $\rangle$

$$\mathrm{Dep\_invalid}(ts_1) \wedge \mathrm{Dep\_invalid}(ts_2)$$

□

A.2 Proofs of the Results of Chapter 6

A.2.1 Termination of the Rewriting System

Lemma A.11. *We have* $(l >_{lop} r)$ *for the rules* $\big(\{x\}, +(x,0) \longrightarrow x\big)$, $\big(\{x\}, +(x,x) \longrightarrow x\big)$, *and* $\big(\{x\}, \cdot(x,1) \longrightarrow x\big)$.

Proof.

$l >_{lop} r$

$\Longleftrightarrow$ $\langle$ Definition of $>_{lpo}$: LOP1. Particularly, $x \in \mathcal{V}ar(+(x,0)) \wedge x \neq +(x,0)$, $x \in \mathcal{V}ar(+(x,x)) \wedge x \neq +(x,x)$, and $x \in \mathcal{V}ar(\cdot(x,1)) \wedge x \neq \cdot(x,1)$ $\rangle$

true

□

Lemma A.12. *We have* $(l >_{lop} r)$ *for the rewriting rule* $\big(\{x,y,z\}, \cdot(x, +(y,z)) \rightarrow +(\cdot(x,y), \cdot(x,z))\big)$.

Proof. For the sake of clarity, we divide our proof into 3 parts as follows:

1. We first prove $\cdot(x, +(y,z)) >_{lop} x, y, z$, and $+(y,z) >_{lop} y, z$.

$\big(+(y,z) >_{lop} y, z\big) \wedge \big(\cdot(x, +(y,z)) >_{lop} x, y, z\big)$

$\Longleftrightarrow$ $\langle$ Definition of $>_{lpo}$: LOP1. In particular, we have $y, z \in \mathcal{V}ar\big(+(y,z)\big) \wedge y, z \neq +(y,z)$, and $x, y, z \in \mathcal{V}al\big(\cdot(x, +(y,z))\big) \wedge x, y, z \neq \cdot(x, +(y,z))$ $\rangle$

true

2. We prove $\cdot(x, +(y, z)) >_{lop} \cdot(x, y)$ and $\cdot(x, +(y, z)) >_{lop} \cdot(x, z)$.

   Proof (1): $\big(\cdot(x, +(y, z)) >_{lop} x\big) \wedge \big(\cdot(x, +(y, z)) >_{lop} y\big) \wedge \big(+(y, z) >_{lop} y\big)$

   $\Longleftrightarrow$ $\langle$ Definition of $>_{lpo}$: LOP2c. In particular, let $f = g = \cdot$, $s = \cdot(x, +(y, z))$, $s_1 = x$, $s_2 = +(y, z)$, $t = \cdot(x, y)$, $t_1 = x$, $t_2 = y$, and $n = m = 2$. $\rangle$

   $\cdot(x, +(y, z)) >_{lop} \cdot(x, y)$

   Since the proof for $\cdot(x, +(y, z)) >_{lop} \cdot(x, z)$ is quite similar with the above proof, we omit the detail here.

3. Finally, we prove $\cdot(x, +(y, z)) >_{lop} +(\cdot(x, y), \cdot(x, z))$.

   Proof (2): $\big(\cdot(x, +(y, z)) >_{lop} \cdot(x, y)\big) \wedge \big(\cdot(x, +(y, z)) >_{lop} \cdot(x, z)\big)$

   $\Longleftrightarrow$ $\langle$ Definition of $>_{lpo}$: LOP2b In particular, $f = \cdot > g = +$, $s = \cdot(x, +(y, z))$, $t_1 = \cdot(x, y)$, $t_2 = \cdot(x, z))$, and $n = 2$. $\rangle$

   $\cdot(x, +(y, z)) >_{lop} +(\cdot(x, y), \cdot(x, z))$.

□

Lemma A.13. *We have* $(l >_{lop} r)$ *for the rewriting rule* $\big(\{x\}, \cdot(x, 0) \longrightarrow 0\big)$.

Proof.

$(+ > 0) \wedge (0 \geq 0)$

$\Longleftrightarrow$ $\langle$ Definition of $>_{lpo}$: LOP2a. In particularly, $f = +$, $g = 0$, $s = +(x, 0)$, $s_1 = x$, $s_2 = 0$, $t = 0$, and $m = 2$. $\rangle$

$+(x, 0) >_{lop} 0$

□

Lemma A.14. *We have* $(l >_{lop} r)$ *for any rule in* $R(E_{spec})$.

Proof.

$$
\begin{array}{ll}
& \mathsf{true} \\
\Longrightarrow & \langle \text{ The syntactical requirements of PFA specification } \rangle \\
& \forall((X, L, R) \in Eq(S) \ |: \ \exists(0 \leq i \leq m \ |: \ L = F_i \wedge \ \mathcal{V}al(R) = X - \{F_i\} \\
& \qquad\qquad \wedge \mathcal{V}al(R) \subseteq \{f_1, \ldots f_n, F_1, \ldots, F_{i-1}\}\,)\,) \\
\Longleftrightarrow & \langle \text{ Definition of } R(E_{spec})\text{: Expression (6.21) } \rangle \\
& \forall((X, l \longrightarrow r) \in R(E_{spec}) \ |: \ \exists(0 \leq i \leq m \ |: \ l = F_i \wedge \ \mathcal{V}al(r) = X - \{F_i\} \\
& \qquad\qquad \wedge \mathcal{V}al(r) \subseteq \{f_1, \ldots f_n, F_1, \ldots, F_{i-1}\}\,)\,) \\
\Longleftrightarrow & \langle \text{ Use the Definition of } >_{lpo}\text{: LOP2b recursively. In particular,} \\
& \quad s = F_i > \cdot, +, \text{ and } (F_i > f_1) \wedge \cdots \wedge (F_i > f_n) \wedge (F_i > F_1) \wedge \cdots \wedge \\
& \quad (F_i > F_{i-1}) \wedge (F_i > 1) \wedge (F_i > 0) \ \rangle \\
& \forall((X, L \longrightarrow R) \in R(E_{spec}) \ |: \ l >_{lop} r\,)
\end{array}
$$

□

Theorem A.15. *For any syntactically correct base specification* S *and aspect specification* A*, the rewriting system* $R(E_f) \cup R(E_{spec})$ *is terminating.*

Proof.

The rewriting system $R(E_f) \cup R(E_{spec})$ is given by Table 6.2 and Expression (6.21)

$\Longleftrightarrow$ $\langle$ Lemmas A.11– A.14 $\rangle$

$$\forall((X, l \longrightarrow r) \mid (X, l \longrightarrow r) \in R(E_f) \cup R(E_{spec}) : l >_{lop} r)$$

$\Longleftrightarrow$ $\langle$ Theorem 3.5 and Theorem 3.6, which indicate $>_{lop}$ is a reduction order & Theorem 3.7 $\rangle$

The rewriting system $R(E_f) \cup R(E_{spec})$ is terminating.

□

A.2.2 Confluence of the Rewriting System

Lemma A.16. *Let* $U = \{(a, b) \mid a, b \in R(E_f) \cup R(E_{spec})\}$. *The union of the following subset of* U *is* U.

S_1 : *Any two distinct rewriting rules from* $R(E_{spec})$

S_2 : *Any one rewriting rule from* $R(E_{spec})$, *and any one rule in* $R(E_f)$

S_3 : *Any one rewriting rule of* $r3$ *and* $r4$, *and any rule of* $r7$ *and* $r9$.

S_4 : *The rule* $r3$ *and the rule* $r4$.

S_5 : *The rule* $r7$ *and the rule* $r9$.

S_6 : *The rule* $r8$ *and any one rule of* $r7$ *and* $r9$.

S_7 : *The rule* $r8$ *and the rule* $r3$.

S_8 : *The rule* $r8$ *and the rule* $r4$.

S_9 : *Any one rule of* $R(E_f) \cup R(E_{spec})$ *with itself.*

Proof. $S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5 \cup S_6 \cup S_7 \cup S_8 \cup S_9$

$\Longleftrightarrow$ $\langle$ Definition of S_6, S_7, S_8, and E_f & idempotent, commutativity and associativity of $\cup$ $\rangle$

$(S_1 \cup S_9) \cup S_2 \cup (S_3 \cup S_4 \cup S_5 \cup S_9)$

$\cup\{(a,b) \mid (a \in \{r8\} \wedge b \in E_f - \{r8\}) \vee (a \in E_f - \{r8\} \wedge b \in \{r8\})\}$

$\Longleftrightarrow$ $\langle$ Definition of S_3, S_4, S_5, S_9, and E_f $\rangle$

$(S_1 \cup S_9) \cup S_2 \cup \{(a,b) \mid a \in E_f - \{r8\} \wedge b \in E_f - \{r8\}\}$

$\cup\{(a,b) \mid (a \in \{r8\} \wedge b \in E_f - \{r8\}) \vee (a \in E_f - \{r8\} \wedge b \in \{r8\})\}$

$\cup\{(a,b) \mid (a \in \{r8\} \wedge b \in \{r8\}\}$

$\Longleftrightarrow$ $\langle$ Definition of set union & Definition of E_f $\rangle$

$(S_1 \cup S_9) \cup S_2 \cup \{(a,b) \mid a,b \in E_f\}$

$\Longleftrightarrow$ $\langle$ Definitions of S_1, S_2 and S_9 $\rangle$

$\{(a,b) \mid a,b \in Eq\} \cup \{(a,b) \mid (a \in Eq \wedge b \in E_f) \vee (a \in E_f \wedge b \in Eq)\}$

$\cup\{(a,b) \mid a,b \in E_f\}$

$\Longleftrightarrow$ $\langle$ Definition of set union $\rangle$

$\{(a,b) \mid a,b \in Eq \cup E_f\}$

□

Lemma A.17. *Any two distinct rewriting rules in $R(E_{spec})$ cannot give raise of a critical pair.*

Proof. For the case of redundant specification, (i.e., two rewrite rules generated by equations of E_{spec} are the same), the possibility of giving raise of critical pair is the same as the case for the set of S_9. We will discuss this case later. In the following

proof, we assume that all rewrite rules in $R(E_{spec})$ are different. According to the definition of $R(E_{spec})$, we have:

$$\forall(l_1 \to r_1, l_2 \to r_2 \in R(E_{spec}) \mid l_1 \neq l_2 : l_1, l_2 \in \mathfrak{L}(S))$$

$\Longleftrightarrow$ $\langle$ Lemma 3.8. In particular, there is no mgu for two distinct constants $\rangle$

$$\forall(l_1 \to r_1, l_2 \to r_2 \in R(E_{spec}) \mid l_1 \neq l_2 : \neg\, \exists(\theta, p \mid: \theta(l_1|_p) = \theta l_2))$$

$\Longleftrightarrow$ $\langle$ Definition 3.27 of critical pair $\rangle$

$\forall(l_1 \to r_1, l_2 \to r_2 \in R(E_{spec}) \mid l_1 \neq l_2 : l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair)

□

Lemma A.18. *Any one rewriting rule in $R(E_{spec})$ with any one rule in $R(E_f)$ cannot give raise of a critical pair.*

Proof. To show that any one rewriting rule in $R(E_{spec})$ with any one rule in $R(E_f)$ cannot give raise a critical pair, we divide proof into two parts.

(1) In the first part, we assume $l_1 \to r_1 \in R(E_{spec})$ and $l_2 \to r_2 \in R(E_f)$. According to the definition of $R(E_{spec})$ and $R(E_f)$, we have:

$$\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_{spec}), l_2 \to r_2 \in R(E_f) : l_1 \in \mathfrak{L}(S) \wedge l_2 \text{ is a non-variable and non-constant term})$$

$\Longleftrightarrow$ $\langle$ Lemma 3.8. In particular, there is no mgu for a constant with a non-variable and non-constant term $\rangle$

$$\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_{spec}) \wedge l_2 \to r_2 \in R(E_f) : \neg\, \exists(\theta, p \mid: \theta(l_1|_p) = \theta l_2))$$

$\Longleftrightarrow$ $\langle$ Definition 3.27 of critical pair $\rangle$

$$\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_{spec}) \wedge l_2 \to r_2 \in R(E_f) \ : \ l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ cannot give raise of critical pair})$$

(2) In the second part, we assume $l_1 \to r_1 \in R(E_f)$ and $l_2 \to r_2 \in R(E_{spec})$. According to the definition of $R(E_{spec})$ and $R(E_f)$, we have

$$\begin{aligned}
&\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_f) \wedge l_2 \to r_2 \in R(E_{spec}) \ : \ l_2 \in \mathfrak{L}(S) \wedge \text{ the non-variable subterm of } l_1 \text{ is either a constant 0 or 1, or } l_1 \text{ itself} \ ) \\
\iff \quad &\langle \text{ Lemma 3.8. In particular, there is no mgu for either two distinct constants, or a constant with a non-variable and non-constant term. } \rangle \\
&\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_f) \wedge l_2 \to r_2 \in R(E_{spec}) \ : \ \neg \exists(\theta, p \mid \theta(l_1|_p) \text{ is not a variable } : \ \theta(l_1|_p) = \theta l_2 \,)\,) \\
\iff \quad &\langle \text{ Definition 3.27 of critical pair } \rangle \\
&\forall(l_1, l_2 \mid l_1 \to r_1 \in R(E_f) \wedge l_2 \to r_2 \in R(E_{spec}) \ : \ l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ cannot give raise of critical pair})
\end{aligned}$$

□

Lemma A.19. *Any rewriting rule among* $r3$ *and* $r4$*, and any rewriting rule among* $r7$ *and* $r9$ *cannot give raise of a critical pair.*

Proof. To show that any one rewriting rule of $r3$ and $r4$, and any one rewriting rule $r7$ and $r9$ cannot give raise of a critical pair, we divide the proof into two parts.

(1) In the first part, we assume that $l_1 \to r_1$ is either $r3$ or $r4$, while $l_2 \to r_2$ is either $r7$ or $r9$. According to the definitions of $r3$, $r4$, $r7$, and $r9$, we have

$$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r3, r4, \} \wedge l_2 \to r_2 \in \{r7, r9\} \ : \ \text{ the non-variable}$$

subterm of l_1 is either a constant 0 or 1, or l_1 itself, which is in the form of $+(t_1, t_2) \wedge l_2$ is a term in the form of $\cdot(t_1', t_2')$)

$\Longleftrightarrow$ ⟨ Lemma 3.8. In particular, there is no mgu for a constant with a non-variable and non-constant term, and there is no mgu for two terms in the form of $+(t_1, t_2)$ and $\cdot(t_1', t_2')$ ⟩

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r3, r4, \} \wedge l_2 \to r_2 \in \{r7, r9\} : \neg \exists(\theta, p \mid \theta(l_1|_p)$ is not a variable $: \theta(l_1|_p) = \theta l_2))$

$\Longleftrightarrow$ ⟨ Definition 3.27 of critical pair ⟩

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r3, r4, \} \wedge l_2 \to r_2 \in \{r7, r9\} : l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair)

(2) In the second part, we assume that $l_1 \to r_1$ is either $r7$ or $r9$, while $l_2 \to r_2$ is either $r3$ or $r4$. According to the definitions of $r3$, $r4$, $r7$, and $r9$, we have

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r3, r4, \} :$ the non-variable subterm of l_1 is either a constant 0 or 1, or l_1 itself, which is in the form of $\cdot(t_1, t_2) \wedge l_2$ is a term in the form of $+(t_1', t_2'))$

$\Longleftrightarrow$ ⟨ Lemma 3.8. In particular, there is no mgu for a constant with a non-variable and non-constant term, and there is no mgu for two terms in the form of $+(t_1, t_2)$ and $\cdot(t_1', t_2')$ ⟩

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r3, r4, \} : \neg \exists(\theta, p \mid \theta(l_1|_p)$ is not a variable $: \theta(l_1|_p) = \theta(l_2)))$

$\Longleftrightarrow$ ⟨ Definition 3.27 of critical pair ⟩

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r3, r4, \} : l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair)

□

Lemma A.20. *The critical pairs determined by the rewriting rule $r3$ and $r4$ are joinable.*

Proof. The rules $r3$ and $r4$ can give raise of two critical pairs. We respectively show that both critical pairs are joinable.

(1) Assume that $r3$ is $l_1 \to r_1$ and $r4$ is $l_2 \to r_2$. According to definitions of $r3$ and $r4$, we have:

$$
\begin{aligned}
& l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r4\} \implies \text{the non-variable subterm of } l_1 \text{ is} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{either } 0, \text{ or } +(x,0) \wedge l_2 = +(x',x') \\
& \qquad \langle \text{ Definition of mgu. In particular, } \theta = \{x \mapsto x', x' \mapsto 0\} \rangle \\
& l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r4\} \implies \exists(\theta, p \ |: \ \theta(l_1|_p) = \theta(l_2) = 0 + 0\,) \\
\iff & \qquad \langle \text{ Definition 3.27 of critical pair} \quad \& \quad \theta = \{y \mapsto x', z \mapsto 0\} \rangle \\
& l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r4\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of a} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{critical pair} \langle x', x' \rangle \\
\iff & \qquad \langle \ x \downarrow x \ \rangle \\
& l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r4\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of a} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{joinable critical pair.}
\end{aligned}
$$

(2) Assume that $r4$ is $l_1 \to r_1$ and $r3$ is $l_2 \to r_2$. According to definitions of $r3$ and $r4$, we have:

$$
\begin{aligned}
& l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r3\} \implies \text{the non-variable subterm of } l_1 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{is } +(x,x) \wedge l_2 = +(x',0) \\
& \qquad \langle \text{ Definition of mgu. In particular, } \theta = \{x' \mapsto x, x \mapsto 0\} \rangle \\
& l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r3\} \implies \exists(\theta, p \ |: \ \theta(l_1)|_p = \theta(l_2) = +(0,0)\,)
\end{aligned}
$$

$\iff$ $\langle$ Definition of critical pair 3.27 & $\theta = \{x \mapsto x, x \mapsto 0\}$ $\rangle$

$l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r3\} \implies l_1 \to r_1$ and $l_2 \to r_2$ give raise of a critical pair$\langle 0, 0\rangle$

$\iff$ $\langle$ $0 \downarrow 0$ $\rangle$

$l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r3\} \implies l_1 \to r_1$ and $l_2 \to r_2$ give raise of a joinable critical pair.

□

Lemma A.21. *The rewriting rule $r7$ and $r9$ cannot give raise of a critical pair.*

Proof. We also consider the proof in two cases.

(1) Assume that $r7$ is $l_1 \to r_1$ and $r9$ is $l_2 \to r_2$. According to definitions of $r7$ and $r9$, we have:

$l_1 \to r_1 \in \{r7\} \wedge l_2 \to r_2 \in \{r9\} \implies$ the non-variable subterm of l_1 is either 1, or $\cdot(x, 1) \wedge l_2 = \cdot(x', 0)$

$\langle$ Lemma 3.8. In particular, there is no unifier for two distinct constants, and there is no unifier for a constant with a non-variable and non-constant term. $\rangle$

$l_1 \to r_1 \in \{r7\} \wedge l_2 \to r_2 \in \{r9\} \implies \neg\, \exists(\theta, p \ |: \ \theta(l_1)|_p = \theta(l_2))$

$\iff$ $\langle$ Definition 3.27 of critical pair $\rangle$

$l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r4\} \implies l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair

(2) Assume that $r9$ is $l_1 \to r_1$ and $r7$ is $l_2 \to r_2$. According to definitions of $r7$ and $r9$, we have:

$l_1 \to r_1 \in \{r9\} \wedge l_2 \to r_2 \in \{r7\} \implies$ the non-variable subterm of is either l_1 0, or $\cdot(x, 0) \wedge l_2 = \cdot(x', 1)$

$\langle$ Lemma 3.8. In particular, there is no unifier for two distinct constants, and there is no unifier for a constant and a non-variable and non-constant term $\rangle$

$l_1 \to r_1 \in \{r9\} \wedge l_2 \to r_2 \in \{r7\} \implies \neg\, \exists(\theta, p \mid: \theta(l_1)|_p = \theta(l_2))$

$\iff$ $\langle$ Definition 3.27 of critical pair $\rangle$

$l_1 \to r_1 \in \{r7\} \wedge l_2 \to r_2 \in \{r9\} \implies l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair

□

Lemma A.22. *The rewriting rule $r8$ cannot give raise of critical pairs with $r7$ and $r9$.*

Proof. We also consider the proof in two cases.

(1) Assume that either $r7$ or $r9$ is $l_1 \to r_1$, and $r8$ is $l_2 \to r_2$. According to definitions of $r7$, $r8$, and $r9$, we have:

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r8\} :$ the non-variable strict subterm of l_1 is a constant $\wedge\, l_2$ is a non-variable term)

$\iff$ $\langle$ Lemma 3.8. In particular, there is no unification for a constant and a non-variable term. $\rangle$

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r8\} : \neg\, \exists(\theta, p \mid: \theta(l_1|_p) = \theta l_2))$

$\iff$ $\langle$ Definition 3.27 of critical pair $\rangle$

$\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r7, r9\} \wedge l_2 \to r_2 \in \{r8\} : l_1 \to r_1$ and $l_2 \to r_2$ cannot give raise of a critical pair)

(2) Assume that $r8$ is $l_1 \to r_1$, and either $r7$ or $r9$ is $l_2 \to r_2$. According to definitions of $r7$, $r8$, and $r9$, we have:

$$\begin{aligned}
&\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r7, r9\} : \text{the non-variable strict} \\
&\qquad \text{subterm of } l_1 \text{ is a term in the form of } +(t_1, t_2) \wedge l_2 \text{ is a term} \\
&\qquad \text{in the form of } \cdot(t_1', t_2')\,) \\
\iff &\quad \langle \text{ Lemma 3.8. In particular, there is no unification for two term} \\
&\qquad +(t_1, t_2) \text{ and } \cdot(t_1', t_2') \,\rangle \\
&\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r7, r9\} : \exists(\theta, p \mid: \theta(l_1|_p) = \theta(l_2))\,) \\
\iff &\quad \langle \text{ Definition 3.27 of critical pair } \rangle \\
&\forall(l_1, l_2 \mid l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r7, r9\} : l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ cannot} \\
&\qquad \text{give raise of a critical pair })
\end{aligned}$$

□

Lemma A.23. *The rewriting rule* $r8$ *and* $r3$ *only give raise of one joinable critical pair.*

Proof. We give the proof by two cases.

(1) Assume that $r3$ is $l_1 \to r_1$, and $r8$ is $l_2 \to r_2$. According to definitions of $r8$ and $r3$, we have:

$$\begin{aligned}
&l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r8\} \implies \text{there is no non-variable strict} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{subterm for } l_1 \\
\iff &\quad \langle \text{ Definition of critical pair 3.27 } \rangle \\
&l_1 \to r_1 \in \{r3\} \wedge l_2 \to r_2 \in \{r8\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ cannot} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{give raise of a critical pair}
\end{aligned}$$

(2) Assume that $r8$ is $l_1 \to r_1$, and $r3$ is $l_2 \to r_2$. According to definitions of $r8$ and $r3$, we have:

$$
\begin{array}{rl}
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r3\} \implies \text{the non-variable strict subterm of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad l_1 \text{ is } +(y,z) \wedge l_2 = +(x',0) \\
& \qquad \langle \text{ Definition of mgu. In particular, } \theta = \{y \mapsto x', z \mapsto 0\} \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r4\} \implies \exists(\theta, p \mid: \theta(l_1)|_p = \theta(l_2) = x' + 0\,) \\
\iff & \qquad \langle \text{ Definition of critical pair 3.27} \quad \& \quad \theta = \{y \mapsto x', z \mapsto 0\} \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r3\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of a} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{critical pair} \langle +(\cdot(x,x'),\cdot(x,0)), \cdot(x,x') \rangle \\
\iff & \qquad \langle\ +(\cdot(x,x'),\cdot(x,0)) \underset{r9}{\longrightarrow} +(\cdot(x,x'),0) \underset{r3}{\longrightarrow} \cdot(x,x')\ \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r3\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{joinable critical pair in the rewriting} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{system } R(E_f) \cup R(E_{spec}).
\end{array}
$$

□

Lemma A.24. *The rewriting rule $r8$ and $r4$ only give raise of one joinable critical pair.*

Proof. We give the proof by two cases.

(1) Assume that $r4$ is $l_1 \to r_1$, and $r8$ is $l_2 \to r_2$. According to definitions of $r8$ and $r4$, we have:

$$
\begin{array}{rl}
& l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r8\} \implies \text{there is no non-variable strict} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{subterm for } l_1 \\
\iff & \qquad \langle \text{ Definition 3.27 of critical pair } \rangle
\end{array}
$$

$$l_1 \to r_1 \in \{r4\} \wedge l_2 \to r_2 \in \{r8\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ cannot give raise of a critical pair.}$$

(2) Assume that $r8$ is $l_1 \to r_1$, and $r4$ is $l_2 \to r_2$. According to definitions of $r8$ and $r4$, we have:

$$\begin{aligned}
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r4\} \implies \text{the non-variable strict subterm of } l_1 \text{ is } +(y,z) \wedge l_2 = +(x',x') \\
& \quad \langle \text{ Definition of mgu. In particular, } \theta = \{y \mapsto x', z \mapsto x'\} \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r4\} \implies \exists(\theta, p \mid: \theta(l_1)|_p = \theta(l_2) = x' + x') \\
\iff & \quad \langle \text{ Definition 3.27 of critical pair } \quad \& \quad \theta = \{y \mapsto x', z \mapsto x'\} \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r4\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of a critical pair } \langle +(\cdot(x,x'),\cdot(x,x')),\cdot(x,x')\rangle \\
\iff & \quad \langle \ +(\cdot(x,x'),\cdot(x,x')) \underset{r4}{\longrightarrow} \cdot(x,x') \ \rangle \\
& l_1 \to r_1 \in \{r8\} \wedge l_2 \to r_2 \in \{r4\} \implies l_1 \to r_1 \text{ and } l_2 \to r_2 \text{ give raise of a joinable critical pair in the rewriting system } R(E_f) \cup R(E_{spec}). \qquad \square
\end{aligned}$$

Lemma A.25. *Any rewriting rules in $R(E_f) \cup R(Eq)$ cannot give raise of a critical pair with itself.*

Proof. To show that any rewriting rules in $R(E_f) \cup R(Eq)$ cannot give raise of a critical pair with itself, we divide the proof into three cases.

(1) Consider rules in $R(E_{spec})$ and the rule r_4. According to their definitions, we have:

$$\forall(l \mid l \to r \in R(E_{spec}) \cup \{r4\} : \text{there is no non-variable proper substerm for } l \ )$$

$\Longleftrightarrow$ $\langle$ Definition of critical pair 3.27. It is safe to ignore $p = \epsilon$ when considering the overlap of one rule itself. $\rangle$

$\forall(l \mid l \to r \in R(E_{spec}) \cup \{r4\} : l \to r$cannot give raise of a critical pair with itself)

(2) Consider the rules $r3$, $r7$ and $r9$. According to their definitions, we have:

$\forall(l \mid l \to r \in \{r3, r7, r9)\}$: the proper non-variable subterm of l is a constant)

$\Longleftrightarrow$ $\langle$ Lemma 3.8. In particular, there is no unifier for a constant and a non-variable and non-constant term $\rangle$

$\forall(l \mid l \to r \in \{r3, r7, r9)\} : \neg \exists(\theta, p \mid p \neq \epsilon : \theta(l)|_p = \theta(l)))$

$\Longleftrightarrow$ $\langle$ Definition of critical pair 3.27. It is safe to ignore $p = \epsilon$ when considering the overlap of one rule itself. $\rangle$

$\forall(l \mid l \to r \in \{r3, r7, r9)\}$: $l \to r$ cannot give raise of a critical pair with itself)

(3) Consider the rule $r8$. According to its definition, we have:

$l \to r \in \{r8)\} \implies$ the proper non-variable subterm of l is in a form of $+(t_1, t_2)$

$\Longleftrightarrow$ $\langle$ Lemma 3.8. In particular, there is no unification for two term $+(t_1, t_2)$ and $\cdot(t'_1, t'_2)$ $\rangle$

$l \to r \in \{r8)\} \implies \neg \exists(\theta, p \mid p \neq \epsilon : \theta(l)|_p = \theta(l))$

$\Longleftrightarrow$ $\langle$ Definition of critical pair 3.27. It is safe to ignore $p = \epsilon$ when considering the overlap of one rule itself. $\rangle$

$l \to r \in \{r8)\} \implies l \to r$ cannot give raise of a critical pair with itself

□

Theorem A.26. *For any syntactically correct specification S and aspect specification A, the rewriting systems $R(E_f) \cup R(E_{spec})$ is confluent.*

Proof. According to the definitions of the rewriting system $R(E_f) \cup R(E_{spec})$, we have:

The rewriting system $R(E_f) \cup R(E_{spec})$ is confluent

$\Longleftrightarrow$ $\langle$ Theorem 6.1 & Theorem 3.9 $\rangle$

All critical pairs of $R(E_f) \cup R(E_{spec})$ are joinable

$\Longleftrightarrow$ $\langle$ Lemmas A.16–A.25 $\rangle$

true

□

A.2.3 Restriction on the selected join points

Lemma A.27. *Given a PFA specification S, let s be a non-ground term in $T_f(\mathfrak{L}(S))$. We have:*

$$\big(R(E_f) \cup R(E_{spec})\big) \vdash (s \downarrow_{AC} = \sum_{i=1}^{m} s_i),$$

where all s_i are not equivalent (up to AC theory of $\cdot$), and each $s_i (1 \leq i \leq m)$ is either 1 or $\prod x_{ij}$ that $x_{ij} \in \mathfrak{L}(S)$.

Proof. Let $s \xrightarrow[R(E_{spec})]{*} s'$, where s' cannot be further rewritten by any rule from $R(E_{spec})$. In other words, no label from $\mathfrak{L}(S)$ in s' matches the left-hand side of any rule in $R(E_{spec})$. Consequently, any term derived from s' by applying rules in $R(E_f)$ also cannot be further rewritten by any rule from $R(E_{spec})$, since applying

any rule in $R(E_f)$ will not create new labels. Hence, we can obtain the AC normal form of s by only applying the AC theory and rules of $R(E_f)$ on s'. In particular, we give the proof based on the inductive definition of s', which is a term in the set $T_f(\mathfrak{L}(S)) - T_f(\emptyset)$.

The strict subterm relationship $\prec$ gives a well-founded order over the set of terms $T_f(\mathfrak{L}(S)) - T_f(\emptyset)$. Let us define the structural inductive property formally as follows:

$$P(s') \iff \big(R(E_f)\big) \vdash (s' \downarrow_{AC} = \sum_{i=1}^{m} s_i),$$

where all s_i are not equivalent (up to AC theory of $\cdot$), and each s_i is either 1 or $\prod x_{ij}$ that $x_{ij} \in \mathfrak{L}(S)$.

1. **Base cases:** The minimal elements in $T_f(\mathfrak{L}(S)) - T_f(\emptyset)$ are all labels in $\mathfrak{L}(S)$. We need to prove $\forall(x \mid x \in \mathfrak{L}(S) : P(x) \iff \big(R(E_f)\big) \vdash (x \downarrow_{AC} = \sum_{i=1}^{m} s_i))$.

   The proof is trivial. Any label x from $\mathfrak{L}(S)$ can be considered as a case of $\sum_{i=1}^{m} s_i$, where $m = 1$ and $s_1 = x$. Furthermore, no axioms of AC theory and rules of $R(E_f)$ can be applied to x, since a label from $\mathfrak{L}(S)$ cannot be an instance of any term at the left-hand sides of the rewriting rules of $R(E_f)$.

2. **Inductive cases:** All constructs over terms are $+$ and $\cdot$. We have $t_1, t_2 \prec t_1 + t_2$, and $t_1, t_2 \prec t_1 \cdot t_2$. Let assume that:

   $P(t_1) \iff \big(R(E_f)\big) \vdash t_1 \downarrow_{AC} = \sum_{i=1}^{m_1} p_i$, where all p_i are not equivalent (up to AC theory of $\cdot$), and each $p_i (1 \leq i \leq m)$ is either 1 or $\prod x_{ij}$ that $x_{ij} \in \mathfrak{L}(S)$.

   $P(t_2) \iff \big(R(E_f)\big) \vdash t_2 \downarrow_{AC} = \sum_{i=1}^{m_2} q_i$, where all q_i are not equivalent (up

to AC theory of $\cdot$), and each $q_i (1 \leq i \leq m)$ is either 1 or $\prod x_{ij}$ that $x_{ij} \in \mathfrak{L}(S)$.

(a) We need to prove $P(t_1) \wedge P(t_2) \implies P(t_1 + t_2)$. The idea is to obtain the AC normal form of $t_1 + t_2$ according to the AC normal form of t_1 and t_2.

$t_1 + t_2$

$\xrightarrow[R(E_f)]{*_{AC}}$ $\langle$ Assumption: the AC normal form of t_1 and t_2 $\rangle$

$\sum_{i=1}^{m_1} p_i + \sum_{i=1}^{m_2} q_i$

$\overset{\text{AC}}{=}$ $\langle$ The commutativity of $+$ implies that the sum of a sequence of terms is equivalent (up to the AC theory of $+$) with the sum of a permutation of those terms. Therefore, let $\{p'_1, \ldots, p'_{m_1}\}$ be a permutation of $\{p_1, \ldots, p_{m_1}\}$, and $\{q'_1, \ldots, q'_{m_2}\}$ be a permutation of $\{q_1, \ldots, q_{m_2}\}$ such that $\forall(i \mid 1 \leq i \leq k : p'_{m_1-k+i} \doteq q'_i)$. Assume that k is the number of equivalent (up to the AC theory of $\cdot$) elements of $\{p_1, \ldots, p_{m_1}\}$ and $\{q_1, \ldots, q_{m_2}\}$. $\rangle$

$\sum_{i=1}^{m_1-k} p'_i + (\sum_{i=m_1-k+1}^{m_1} p'_i + \sum_{i=1}^{k} q'_i) + \sum_{i=k+1}^{m_2} q'_i$

$\overset{\text{AC}}{=}$ $\langle$ The commutativity of $\cdot$ also implies that the product of a sequence of labels from $\mathfrak{L}(S)$ is equivalent (up to the AC theory of $\cdot$) with the product of a permutation of those labels. Therefore, let $q''_i (1 \leq i \leq k)$ be a permutation of labels of q'_i such that $p'_{m_1-k+i} = q''_i$. $\rangle$

$\sum_{i=1}^{m_1-k} p'_i + (\sum_{i=m_1-k+1}^{m_1} p'_i + \sum_{i=1}^{k} q''_i) + \sum_{i=k+1}^{m_2} q'_i$

$\xrightarrow[R(E_f)]{*}$ $\langle$ Apply k times the rewrite rule $r4$ for $p'_{m_1-k+i} = q''_i$. $\rangle$

$\sum_{i=1}^{m_1-k} p'_i + \sum_{i=m_1-k+1}^{m_1} p'_i + \sum_{i=k+1}^{m_2} q''_i$

$=$ $\langle$ The definition of $\sum$ $\rangle$

$\sum_{i=1}^{m_1} p'_i + \sum_{i=k+1}^{m_2} q''_i$

$\langle$ No rules of $R(E_f)$ can be further applied to $\sum_{i=1}^{m_1} p'_i + \sum_{i=k+1}^{m_2} q''_i$ or to its equivalent terms (up to the AC theory of $+$ an $\cdot$). $\rangle$

According to the above steps, we obtain the AC normal form of $t_1 + t_2$ as $\sum_{i=1}^{m_1} p'_i + \sum_{i=k+1}^{m_2} q''_i$. Moreover, the assumption of $P(t_1)$ and $P(t_2)$ indicates that each p'_i and q''_i is either 1 or $\prod x_{ij}$ such that $x_{ij} \in \mathfrak{L}(S)$, and all p'_i and q''_i are not equivalent up to the AC theory of $\cdot$. In other words, we prove $P(t_1 + t_2)$, and particularly, $m = m_1 + m_2 - k$, and $t_i = p'_i$ for $1 \leq i \leq m_1$ and $t_i = q''_i$ for $m_1 + 1 \leq i \leq m_1 + m_2 - k$.

(b) We also prove $P(t_1) \wedge P(t_2) \implies P(t_1 \cdot t_2)$ in the same way as the case(a).

$t_1 \cdot t_2$

$\xrightarrow[R(E_f)]{*}_{AC}$ $\langle$ Assumption: the AC normal form of t_1 and t_2. $\rangle$

$\sum_{i=1}^{m_1} p_i \cdot \sum_{i=1}^{m_2} q_i$

$\xrightarrow[R(E_f)]{*}$ $\langle$ Apply the rewrite rule $r8$ $m_1 m_2$ times. $\rangle$

$\sum_{i,j=1}^{m_1,m_2} (p_i \cdot q_j)$

$\xrightarrow[R(E_f)]{*}$ $\langle$ Let $s_{m_2(i-1)+j}$ is $p_i \cdot q_j$ or a term derived from $p_i \cdot q_j$ if $r7$ is applicable $\rangle$

$\sum_{k}^{m_1 m_2} s_k$

$\xrightarrow[R(E_f)]{*}_{AC}$ ⟨ Same as the proof in case (a), we consider a permutation of $s_1, \ldots, s_{m_1 m_2}$ in order to apply the rewrite rules $r4$. In particular, let $s'_1, \ldots, s'_k$ be all non-equivalent (up to the AC theory of $\cdot$) terms from $s_1, \ldots, s_{m_1 m_2}$. ⟩

$\sum_i^k s'_i$

According to the above steps, we obtain the AC normal form of $t_1 + t_2$ as $\sum_{i=1}^k s'_i$, where all s'_i are not equivalent up to the AC theory of $+$ and $\cdot$, and each s'_i is either 1 or $\prod x_{ij}$ such that $x_{ij} \in \mathfrak{L}(S)$. In other word, we prove $P(t_1 \cdot t_2)$, and particularly, $t_i = s'_i$ and $m = k$ such that $1 \leq k \leq m_1 m_2$.

In summary, we prove the $P(s')$ for all $s' \in T_f(\mathfrak{L}(S)) - T_f(\emptyset)$ based on the above proofs of base cases and inductive cases for s'. □

Lemma A.28. *Let s, e be two product family algebra terms such that $\big(R(E_f) \cup R(E_{spec})\big) \vdash \big(s \downharpoonright_{AC} = \sum_{i=1}^m s_i, e \downharpoonright_{AC} = \prod_{i=1}^n y_i\big)$. The term e cannot be extracted from s iff $\forall(i \mid 1 \leq i \leq m : e \downharpoonright_{AC} \nmid s_i)$, where $x \nmid y \iff \neg(x \mid y) \stackrel{\text{def}}{\Leftrightarrow} \neg \exists(z \mid : y = x \cdot z)$.*

Proof. We prove by contradiction.

$\neg \forall(i \mid 1 \leq i \leq m : e \downharpoonright_{AC} \nmid s_i)$

$\iff$ ⟨ Generalized De Morgan ⟩

$\exists(i \mid 1 \leq i \leq m : e \downharpoonright_{AC} \mid s_i)$

$\Longleftrightarrow$ $\langle$ Lemma A.27 & $e \lfloor_{AC} = \prod_{i=1}^{n} y_i$ is a product of labels & Instance of $\exists$. In particular, assume $s_k = e \lfloor_{AC} \cdot t$ where $1 \leq k \leq m$ $\rangle$

$R(E_f) \cup R(E_{spec}) \vdash s \lfloor_{AC} = \sum_{i=1}^{k-1} s_i + e \lfloor_{AC} \cdot t + \sum_{i=k+1}^{m} s_i$

$\Longleftrightarrow$ $\langle$ Associativity and Community of $+$ $\rangle$

$R(E_f) \cup R(E_{spec}) \vdash s \lfloor_{AC} \overset{\text{AC}}{=} e \lfloor_{AC} \cdot t + \sum_{i=1}^{k-1} s_i + \sum_{i=k+1}^{m} s_i$

$\Longleftrightarrow$ $\langle$ Theorem 6.3 $\rangle$

$Th_{pfa} \models (s = e \cdot t + (\sum_{i=1}^{k-1} s_i + \sum_{i=k+1}^{m} s_i))$

$\Longrightarrow$ $\langle$ Assumption: e cannot be extracted from s $\rangle$

false

□

Lemma A.29. *Let e be a term such that $e \lfloor_{AC}$ is a product of labels from $Eq(S)$. Assume β and γ be two terms such that e cannot be extracted from either of them. Assume $\big(R(E_f) \cup R(E_{spec})\big) \vdash \big(\beta \lfloor_{AC} = \sum_{i=1}^{m} p_i\big), \big(\gamma \lfloor_{AC} = \sum_{i=1}^{l} s_i\big)$, then we have*

$$\big(R(E_f) \cup R(E_{spec})\big) \vdash \big((\beta \cdot e + \gamma) \lfloor_{AC} = \sum_{i=1}^{m} (p_i \cdot e \lfloor_{AC}) + \sum_{i=1}^{l} s_i\big).$$

Proof. $\beta \cdot e + \gamma$

$\xrightarrow[R(E_f)]{*}_{AC}$ $\langle$ Assumption of $\beta \lfloor_{AC}$ and $\gamma \lfloor_{AC}$. $\rangle$

$(\sum_{i=1}^{m} p_i) \cdot e + \sum_{i=1}^{l} s_i$

$\xrightarrow[R(E_f) \cup R(E_{spec})]{*}_{AC}$ $\langle$ Definition 6.8 $\rangle$

$(\sum_{i=1}^{m} p_i) \cdot e \lfloor_{AC} + \sum_{i=1}^{l} s_i$

$\xrightarrow[R(E_f)]{*}$ $\langle$ Apply the rewrite rule $r8$ in E_f m times $\rangle$

$\sum_{i=1}^{m}(p_i \cdot e \downarrow_{AC}) + \sum_{i=1}^{l} s_i$

Since $e \downarrow_{AC}$, and all p_i are in the form of products of labels, and all p_i are distinct, we have all $p_i \cdot e \downarrow_{AC}$ are distinct. Moreover, lemma A.28 implies any $p_i \cdot e \downarrow_{AC}$ and any q_i cannot be equivalent. Therefore, no further rules can be applied to any terms that are equivalent to $\sum_{i=1}^{m}(p_i \cdot e \downarrow_{AC}) + \sum_{i=1}^{l} s_i$. □

Lemma A.30. *Given two product family terms s and t, let $\big(R(E_f) \cup R(E_{spec})\big) \vdash \big(s \downarrow_{AC} = \sum_{i=1}^{m} s_i\big)$, and $\big(R(E_f) \cup R(E_{spec})\big) \vdash \big(t \downarrow_{AC} = \sum_{i=1}^{n} t_i\big)$. We have $Th_{pfa} \models s = t$ iff $\exists(f \mid f \in \mathcal{P}(\{1,\ldots,m\}) \times \mathcal{P}(\{1,\ldots,n\}) \wedge f$ is a bijection $: f(i) = j \iff s_i \doteq t_j)$, where $\doteq$ means two terms are equivalent up to the commutativity and associativity of $\cdot$.*

Proof.

$Th_{pfa} \models (s = t)$

$\iff$ $\langle$ Theorem 6.3 $\rangle$

$(R(E_f) \cup R(E_{spec})) \vdash (s \downarrow_{AC} \overset{\text{AC}}{=} t \downarrow_{AC})$

$\iff$ $\langle$ The AC normal form of s and t. $\rangle$

$(R(E_f) \cup R(E_{spec})) \vdash \sum_{i=1}^{m} s_i \overset{\text{AC}}{=} \sum_{i=1}^{n} t_i$

$\iff$ $\langle$ Regarding to associativity and community of $+$ $\rangle$

$\exists(f \mid f \in \mathcal{P}(\{1,\ldots,m\}) \times \mathcal{P}(\{1,\ldots,n\}) \wedge f$ is a bijection

$: f(i) = j \iff s_i \overset{\text{AC}}{=} t_j)$

$\Longleftrightarrow$ $\langle$ Lemma A.27. In particular, all s_i and all t_i are products of labels
& The AC theory of $+$ cannot apply to a term that is a product of labels $\rangle$

$\exists(f \mid f \in \mathcal{P}(\{1,\ldots,m\}) \times \mathcal{P}(\{1,\ldots,n\}) \wedge f$ is a bijection
$: f(i) = j \Longleftrightarrow s_i \doteq t_j)$

□

Theorem A.31. *Let e and p be two given product family terms. Assume that w.r.t. the set of equations Th_{pfa}, $p = \beta_1 \cdot e + \gamma_1 = \beta_2 \cdot e + \gamma_2$, where e cannot be further extracted from β_1, β_2, γ_1 and γ_2. Then if $e \lfloor_{AC}$ is a product of labels, we have $\beta_1 = \beta_2 \wedge \gamma_1 = \gamma_2$ w.r.t. to the set of equations Th_{pfa}.*

Proof. $Th_{pfa} \models \beta_1 \cdot e + \gamma_1 = \beta_2 \cdot e + \gamma_2$

$\Longleftrightarrow$ $\langle$ Theorem 6.3 $\rangle$

$(R(E_f) \cup R(Eq)) \vdash (\beta_1 \cdot e + \gamma_1) \lfloor_{AC} \overset{\text{AC}}{=} (\beta_2 \cdot e + \gamma_2) \lfloor_{AC}$

$\Longleftrightarrow$ $\langle$ Lemma A.29. In particular, let $\beta_1 \lfloor_{AC} = \sum_{i=1}^{m} p_i$, $\beta_2 \lfloor_{AC} = \sum_{i=1}^{n} q_i$, $\gamma_1 \lfloor_{AC} = \sum_{i=1}^{l} s_i$, and $\gamma_2 \lfloor_{AC} = \sum_{i=1}^{r} t_i$. $\rangle$

$(R(E_f) \cup R(Eq)) \vdash \sum_{i=1}^{m}(p_i \cdot e \lfloor_{AC}) + \sum_{i=1}^{l} s_i \overset{\text{AC}}{=} \sum_{i=1}^{n}(q_i \cdot e \lfloor_{AC}) + \sum_{i=1}^{r} t_i$

$\Longleftrightarrow$ $\langle$ Lemma A.28 & Lemma A.30 $\rangle$

$\exists(f \mid f \in \mathcal{P}(\{1,\ldots,m\}) \times \mathcal{P}(\{1,\ldots,n\}) \wedge f$ is a bijection $: f(i) = j$
$\Longleftrightarrow p_i \cdot e \lfloor_{AC} \doteq q_j \cdot e \lfloor_{AC})$
$\wedge \exists(f' \mid f' \in \mathcal{P}(\{1,\ldots,l\}) \times \mathcal{P}(\{1,\ldots,r\}) \wedge f$ is a bijection $: f(i) = j$
$\Longleftrightarrow s_i \doteq t_j)$

$\Longleftrightarrow$ $\langle$ Axiom of $\sum$ & Definition of $\overset{\text{AC}}{=}$ $\rangle$

$(\sum_{i=1}^{m} p_i \overset{\text{AC}}{=} \sum_{i=1}^{n} q_i) \wedge (\sum_{i=1}^{s} s_i \overset{\text{AC}}{=} \sum_{i=1}^{t} t_i)$

$\Longleftrightarrow$ $\langle$ Definitions of $\beta_1 \lfloor_{AC}$, $\beta_2 \lfloor_{AC}$, $\gamma_1 \lfloor_{AC}$, and $\gamma_2 \lfloor_{AC}$ $\rangle$

$(R(E_f) \cup R(E_{spec})) \vdash (\beta_1 \lfloor_{AC} \overset{\text{AC}}{=} \beta_2 \lfloor_{AC}) \wedge (\gamma_1 \lfloor_{AC} \overset{\text{AC}}{=} \gamma_2 \lfloor_{AC})$

$\Longleftrightarrow$ $\langle$ Theorem 6.3 $\rangle$

$Th_{pfa} \models \beta_1 = \beta_2 \wedge \gamma_1 = \gamma_2$

□

Appendix B

Regrading the automation of the weaving process

B.1 Algebraic Specification of AO-PFA Using CASL

library FormalPFA

-++++++++++++++++++++++++++++++++++=

sort *Family*

ops *One* : *Family*;

Zero : *Family*;

$\_\_Choice\_\_ : Family \times Family \rightarrow Family$, *assoc*,

comm, *idem*, *unit Zero*;

$\_\_Mandatory\_\_ : Family \times Family \rightarrow Family$, *assoc*,

comm, *unit One*

end

spec CISemiring[ProductFamily] =

$\forall$ x, y, z : *Family*

• *One Mandatory Zero* = *Zero*

• x *Choice* (y *Mandatory* z)

= (x *Mandatory* y) *Choice* (x *Mandatory* z)

end

spec LABEL =

sort *Alphabet*

free type *Label* ::= [] | __::__(*Alphabet*; *Label*)

sort *Label*

ops *'a'*, *'b'*, *'c'*, *'d'*, *'e'*, *'f'*, *'g'*, *'h'*, *'i'*, *'j'*, *'k'*, *'l'*, *'m'*, *'n'*

: *Alphabet*

ops *'o'*, *'p'*, *'q'*, *'r'*, *'s'*, *'t'*, *'u'*, *'v'*, *'w'*, *'x'*, *'y'*, *'z'* : *Alphabet*

ops *'A'*, *'B'*, *'C'*, *'D'*, *'E'*, *'F'*, *'G'*, *'H'*, *'I'*, *'J'*, *'K'*, *'L'*, *'M'*, *'N'*

: *Alphabet*

ops *'O'*, *'P'*, *'Q'*, *'R'*, *'S'*, *'T'*, *'U'*, *'V'*, *'W'*, *'X'*, *'Y'*, *'Z'*

: *Alphabet*

ops *'0'*, *'1'*, *'2'*, *'3'*, *'4'*, *'5'*, *'6'*, *'7'*, *'8'*, *'9'* : *Alphabet*

op *'_'* : *Alphabet*

op __++__ : *Label* $\times$ *Label* $\rightarrow$ *Label*

pred *null* : *Label*

$\forall$ x, y : *Alphabet*; K, L : *Label*

- $[] ++ K = K$
- $(x :: L) ++ K = x :: (L ++ K)$
- $null(L) \Leftrightarrow L = []$

end

spec MONOID =

sort *Elem*

ops e : *Elem*;

$\_\_*\_\_$: *Elem* $\times$ *Elem* $\rightarrow$ *Elem*, *assoc*, *unit* e

end

view LabelAsMonoid :

MONOID **to** LABEL =

Elem $\mapsto$ *Label*, $e \mapsto []$, $\_\_*\_\_ \mapsto \_\_++\_\_$

end

spec PFA =

LABEL

then sort *Label* $<$ *PFATerm*

op 0 : *PFATerm*

op 1 : *PFATerm*

op $\_\_+\_\_$: *PFATerm* $\times$ *PFATerm* $\rightarrow$ *PFATerm*

op $\_\_*\_\_$: *PFATerm* $\times$ *PFATerm* $\rightarrow$ *PFATerm*

pred *subfamily* : *PFATerm* $\times$ *PFATerm*

pred *refine* : $PFATerm \times PFATerm$

pred *constrain* : $PFATerm \times PFATerm \times PFATerm$

end

spec S1 =

PFA

then **op** *f1* : *Label*

op *f2* : *Label*

op *f3* : *Label*

op *a* : *Label*

• $a = (f1 * f1) + (f2 + f3)$

• *constrain*(*f1*, *a*, *f3*)

end

spec BASE1 =

CISEMIRING

[S1 **fit**

Family $\mapsto$ *PFATerm*, **op** *Zero* $\mapsto$ 0, **op** *One* $\mapsto$ 1,

op __*Choice*__ $\mapsto$ __+__, **op** __*Mandatory*__ $\mapsto$ __*__]

end

library FORMALASPECT

from FORMALPFA **get** PFA

spec ASPECT =

PFA

then sort *Alphabet* < *Alphabetplus*

op *jp* : *Alphabetplus*

free type *Labelplus* ::= [] | __::__(*Alphabetplus*; *Labelplus*)

free type *scope_expr_type* ::= *WITHIN* | *CFLOW*

generated type

scope_expr_conj

::= __{__}(*scope_expr_type*; *Label*)

| __';'__(*scope_expr_conj*; *scope_expr_conj*)

type *scope_expr*

::= [__](*scope_expr_conj*)

| __':'__(*scope_expr*; *scope_expr*)

free type *boolean_expr* ::= *T* | *F*

free type *LIST* ::= *L* | *M* | *R* | *LM* | *MR* | *LR* | *LMR*

free type

kind_type

::= *Declaration*

| *Inclusion*

| *Creation*

| *Component_creation*

| *Component*

| *Equivalent_component*

| *Constraint*(*LIST*)

type $kind\_expr ::= \_\_\{\_\_\}(kind\_type; Label)$

sorts $Labelplus, PFATerm < AspectTerm$

op $jp : AspectTerm$

op $\_\_+\_\_ : AspectTerm \times AspectTerm \to AspectTerm$

op $\_\_*\_\_ : AspectTerm \times AspectTerm \to AspectTerm$

pred $pointcut : scope\_expr \times boolean\_expr \times kind\_expr$

pred $se : scope\_expr$

pred $be : boolean\_expr$

pred $ke : kind\_expr$

end

spec A1 =

ASPECT

then op $x\_jp : Labelplus$

op $f4 : Label$

op $a : Label$

op $b : Label$

• $jp = jp * f4$

if $pointcut([\ WITHIN\ \{\ a\ \}\ ],\ T,\ (Constraint(LR))\ \{\ a\ \})$

end

B.2 Term Rewriting Systems of AO-PFA Using Maude

%#default#.maude includes signature of the commutative idempotent semiring, rewriting rules corresponding to $R(E_f)$, and the AC theory of + and $\cdot$. It won't change w.r.t. to different specifications of base and aspect. %

```
fmod CISR is
sort Family .
ops 0 1 : -> Family .
op _+_ : Family Family  -> Family [prec 33 assoc comm ] .
op _*_ : Family Family -> Family [prec 31 assoc comm ] .
endfm

fmod CISREQ is
including CISR .
vars x y z : Family .
eq x + 0 = x .
eq x + x = x .
eq x * 1 = x .
eq x * (y + z) = x * y + x * z .
eq x * 0 = 0 .
endfm
```

% signature.maude is defined according to a given base specification, and a given aspect specification e.g.,,%

```
load #default#.maude


fmod BASE is
including CISR .
op f1 : -> Family .
op f2 : -> Family .
op f3 : -> Family .
endfm


fmod ASPECT is
including CISR .
op f4 : -> Family .
endfm
```

% trwsystem. maude includes equations corresponding to $R(E_{spec})$, and the rewriting rules defined by the aspects, e.g., %

```
load signature.maude



fmod ADDEDEQ is
including BASE .
eq f3 = f1 + f2 .
```

```
endfm

mod  SUBSTITUTION is
including BASE .
including ASPECT .
var jp : Family .
crl [test] : jp => f4 if jp = f1 * f2 .
endm
```

% #weaving#.maude includes the whole rewriting systems for the weaving process.
%

```
load trwsystem.maude

mod WEAVING is
including CISREQ .
including ADDEDEQ .
including SUBSTITUTION .
endm
```

Bibliography

[AB06] Sven Apel and Don Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the 5th international conference on generative programming and component engineering*, 2006.

[ACLF10] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing feature models. In Mark van den Brand, Dragan Gaševic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin / Heidelberg, 2010.

[AEB03] Omar Aldawud, Tzilla Elrad, and Atef Bader. UML Profile for Aspect Oriented Software Development. In *Workshop on Aspect-Oriented Modeling with UML in AOSD 2003*, 2003.

[AK10] 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.

[ALMK10] 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.

[ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: Aspects and features in concert. In *Proceedings of the International Conference on Software Engineering*, 2006.

[AM03] João Araújo and Ana Moreira. An Aspectual Use Case Driven Approach. In *VIII Jornadas de Ingenierá de Software Bases de Datos*, Alicante, Spain, November 2003.

[ASM+09] 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 *Proceedings of the 2nd International Conference on Software Language Engineering*, 2009.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing Crosscutting Concerns using Composition Filters. *Communications of the ACM*, 44(10):51–57, 2001.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines (SPLC'05)*. Springer-Verlag Berlin, Heidelberg, 2005.

[batmaccRS+05] Ruzanna Chitchyanaspect based approach to modeling access control concerns, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Bedir Tekinerdogan Jethro Bakker, Siobhán Clarke, and Andrew Jackson. Survey of Analysis and Design Approach. Survey, AOSD-Europe, 2005.

[BC04] Elisa Baniassad and Siobhán Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *Proceedings of 26th International Conference on Software Engineering (ICSE 2004)*, pages 158 – 167, May 2004.

[BCD+04] Olivier. Barais, Eric Cariou, Laurence Duchien, Nicolas Pessemier, and Lionel Seinturier. TranSAT: A Framework for the Specification of Software Architecture Evolution. In *The First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Jun 2004.

[Ber94] Lodewijk Bergmans. The composition filters object model. Technical report, Dept. of Computer Science, University of Twentey, 1994.

[BM04] Michel Bidoit and Peter D. Mosses. CASL user manual: Introduction to using the common algebraic specification language. In *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[BMB+10] Marko Bošković, Gunter Mussbacher, Ebrahim Bagheri, Daniel Amyot, Dragan Gaš ević, and Marek Hatala. Aspect-oriented

feature models. In *Position Paper. Proceedings of the 2010 international conference on Models in software engineering*, 2010.

[BMN+06] Johan Brichau, Mira Mezini, Jacques Noyé, Wilke Havinga, Lodewijk Bergmans, Vaidas Gasiunas, Christoph Bockisch, Johan Fabry, and Theo D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. Research, AOSD-Europe, February 2006.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BP06] Matthias Baaz and Norbert Preining. First-order gödel logics. *Analysis of Pure and Applied Logic*, 2006.

[Bra03] Jeremy T. Bradley. An Examination of Aspected Oriented Programming in Industry. Technical report, Colorado State University Honors Program, 2003.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, pages 615–636, 2010.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advance Information Systems Engineering, 17th International Conference*, 2005.

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude*

Manual (Version 2.6). SRI International Computer Science Laboratory, 2011.

[CEW93] Ingo Claβen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic Specification Techniques and Tools For Software Development: The ACT Approach*. World Scientific Publishing Co. Pte. Ltd., 1993.

[CN02] Paul Clements and Linda Northrop. *Software Product Line: Practices and Patterns*. Addison-Wesley, 2002.

[Coh02] Sholom Cohen. Product line state of the practice report. Technical Report CMU/SEI-2002-TN-017, The Software Engineering Institute, September 2002.

[CRB04] Adrian Colyer, Awais Rashid, and Gordon Blair. On the separation of concerns in product families. Technical report, Computing Department, Lancaster University, 2004.

[Cza98] 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.

[Dev03] Devereux. Compositional reasoning about aspects using alternating-time logic. *A Foundations of Aspect Languages (FOAL) Workshop*, 2003.

[Die05] Reinhard Dieste. *Graph Theory*. Springer-Verlag Berlin Heidelberg, 2005.

[EAK+1a] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing Aspect of AOP. *Communications of the ACM*, pages 33–38, 2001a.

[EBB05] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS approach-domain modeling with features, use cases and use realization. In *Proceedings of 9th International Conference on Software Product Lines*, 2005.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Communications of THE ACM*, pages 29–31, 2001.

[EM85] H. EHRIG and B. MAHR. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag Berlin Heidelberg, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specification and Constraints*. Springer-Verlag, 1990.

[FF01] Rober E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, Research Institute for Advanced Computer Science, May 2001. Workshop on Advanced Separation of Concerns (OOPSLA 2000).

[FZ06] Shaofeng Fan and Naixiao Zhang. Feature model based on description logics. In *Proceedings of the 10th international conference on Knowledge-Based Intelligent Information and Engineering Systems - Volume Part II (KES'06)*, 2006.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating features modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, 1998.

[Gri00a] Martin L. Griss. Implementing Product-Line Features By Composing Component Aspects. In *Proceedings of First International Software Product Line Conference*, Denver, CO,, 2000.

[Gri00b] Martin L. Griss. Implementing Product-Line Features with Component Reuse. In *Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability*, pages 137 – 152. Springer-Verlag, Jun 2000.

[GV07] Iris Groher and Markus Voelter. Xweave: Models and aspects in concert. In *Proceedings of the 10th Workshop on Aspect-Oriented Modeling*, 2007.

[Her02] Stephan Herrmann. Composable Designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML in AOSD 2002*, 2002.

[HKM06] 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.

[HKM08] Peter Höfner, Ridha Khedri, and Bernhard Möller. Algebraic view reconciliation. In *Proceedings of 6th IEEE International Conference on Software Engineering and Formal Methods*, 2008.

[HKM11a] Peter Höfner, Ridha Khedri, and Bernhard Möller. An algebra of product families. *Software and Systems Modeling*, 10(2):161–182, 2011.

[HKM11b] Peter Höfner, Ridha Khedri, and Bernhard Möller. Supplementing product families with behaviour. *International Journal of Informatics*, pages 245 – 266, 2011.

[HL95] Walter Hürsch and Cristina Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, February 1995.

[HLM+98] Geoff Hulten, Karl Lieberherr, Josh Marshall, Doug Orleans, and Binoy Samuel. *Demeter/Java: User Manual*. Northeastern University, Boston, MA, 1998.

[HLN04] Charles Haley, Robin Laney, and Bashar Nuseibeh. Deriving security requirements from crosscutting threat description. In *Proceedings of the 3rd International Conferece on Aspect Oriented Software Development (AOSD 04)*. ACM Press, 2004.

[HMM05] Abdel Hakim Hannousse, Djamel Meslati, and Hayette Merouani. Aspect Oriented Programming and Composition Filters: A Conceptual Comparative Study. In *Iberian Workshop on Aspect Oriented Software Development (DSOA'05)*, 2005.

[HO93] William Harrison and Harold Ossher. Subject-Oriented Programming - Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993.

[IC08] Mustafa Ispir and Aysu Betin Can. An assume guarantee verification methodology for aspect oriented programming. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 391–394, 2008.

[JN04] Ivar Jacobson and PanWei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional, 2004.

[Kan03] Mohamed Mancona Kandé. *A concern-oriented approach to software architecture*. PhD thesis, Swiss Federal Institute of Techniques, 2003.

[Kat04] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In *FOAL Workshop, associated with AOSD*, 2004.

[Kat05] Shmuel Katz. A survey of verification and static analsyis for aspect. Technical report, AOSD-Europe, 2005.

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I*, pages 106–134. Springer-Verlag, 2006.

[KBK09] Martin Kuhlemann, Don Batory, and Christian Kästner. Safe composition of non-monotonic features. In *Proceedings of the eighth international conference on Generative programming and component engineering*, pages 177–186, 2009.

[KCH+90] 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.

[KFG04] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137 – 146, 2004.

[KG99] Shmuel Katz and Joseph (Yossi) Gil. Aspects and superimpositions (position paper). In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP 1999*, 1999.

[KGdLvS04] Uirá Kulesza, Alessandro Fabricio Garcia, Carlos José Pereira de Lucena, and Arndt von Staa. Integrating Generative and Aspect-Oriented Technologies. In *19th ACM SIGSoft Brazillian Symposium on Software Engineering*, 2004.

[kKFGS02] Dae kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. Using role-based modeling language (rbml) as precise characterizations of model families. In *In Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2002.

[KKG07] Avadhesh Kumar, Rajesh Kumar, and P.S. Grover. An Evaluation of Maintainability of Aspect-Oriented Systems: A Practical Approach. *International Journal of Computer Science and Security*, 1(2):1–9, 2007.

[KKL+01] 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.

[KLD02] Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[KT09] Jaakko Kuusela and Harri Tuominen. Aspect-Oriented Approach

to Operating System Development Empirical Study. *Journal of Communication and Computer*, 6(8):233–238, 2009.

[Lie96] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[LKF02] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying crosscutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6), 2002.

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques and Tools*, 2002.

[LN10] Jørn Lind-Nielsen. Buddy BDD Library. `http://sourceforge.net/projects/buddy/`, 2010. (Last accessed on March 28, 2013).

[LOO01] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of THE ACM*, 44(10):39–41, 2001.

[LRCe06] Neil Loughran, Awais Rashid, Ruzanna Chitchyan, and etal. A domain analysis of key concerns-known and new candidates. Research, AOSD-Europe, 2006.

[MAR05] Ana Moreira, João Araújo, and Awais Rashid. A Concern-Oriented Requirements Engineering Model. In João Falcão

e Cunha Oscar Pastor, editor, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portuga, 2005.

[MK03] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2003)*, Finland, 2003.

[MNJP02] John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Guest editors' introduction: Initiating software product lines. *IEEE Software*, 19(4):24–27, 2002.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with Casesar. In *Proceedings of conference on aspect-oriented software development*, 2003.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM International Symposium on Foundations of Software Engineering*, 2004.

[MRA05] Ana Moreira, Awais Rashid, and João Araújo. Multi-dimensional separation of concerns in requirements engineering. In *Proceedings of the 13th IEEE International Requirements Engineering Conference (RE2005)*, pages 285–296, 2005.

[MWCC08] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and

Donald Cowan. Efficient compilation techniques for large scale feature models. In *7th international conference on Generative programming and component engineering - GPCE '08*, 2008.

[OSRSC01] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, 2001.

[OT01] Harold Ossher and Peri Tarr. Using Multidimensional Separation of Concerns To (Re) Shape Evolving Software. *Communication of the ACM*, 44(10):43–50, 2001.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:1053–1058, 1972.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering: foundations, principles, and techniques*, chapter 3, pages 39–52. Springer-Verlag, Berlin, Heidelberg, 2005.

[PC01] Andrés Díaz Pace and Marcelo Campo. An Empirical Study About Separation of Concerns Approaches. In *Proceedings of the 2nd Argentine Symposium on Software Engineering (ASSE 2001), 30 th Argentine Conference on Computer Science and Operational Research*, Buenos Aires, 2001.

[PFT03] Mónica Pinto, Lidia Fuentes, and Jose María Troya. DAOP-ADL:

An Architecture Description Languages for Dynamic Component and Aspect-Based Development. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 118 – 137. Springer-Verlag, 2003.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicites. In *Proceedings of 6th Conference on Integrated and Design Process Technology*, 2002.

[RFLG04] Indrakshi Ray, Robert France, Na Li, and Geri Georg. An aspect-based approach to modeling access control concerns. *Information and software technology*, 46(9):575–587, 2004.

[RM06] Awais Rashid and Ana Moreira. Domain models are not aspect free. In *In MODELS*, pages 155–169. Springer, 2006.

[RMA03] 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, Boston, USA, 2003.

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147 – 158, 2004.

[SH04] Viren Shah and Frank Hill. An aspect-oriented security framework: lessons learned. In *Workshop of AOSD Technology for Application-level Security (AOSDSEC)*, 2004.

[SHU02] Dominik Stein, Stefan Hanenherg, and Rainer Unland. A UML-based Aspect-Oriented Design Notation for AspectJ. In *Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 106 – 112. ACM New York, 2002.

[Sip03] Henny B. Sipma. A formal model for cross-cutting modular transition systems. *Foundations of Aspect Languages (FOAL) Workshop associated with AOSD*, 2003.

[SK03] MARCELO SIHMAN and SHMUEL KATZ. sumperimposition and aspect-oriented programming. *The Computer Journal*, 46:529–541, 2003.

[SSK+07] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A survey on Aspect-Oriented Modeling Approaches. Technical report, Vienna University of Technology, 2007.

[Str04] Detlef Streiferdt. *Family-Oriented Requirements Engineering*. PhD thesis, Technical University Ilmenau, IImenau, Germany, 2004.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook.

Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, 2007.

[Tek04] Bedir Tekinerdoğan. ASAAM: Aspectual Software Architecture Analysis Method. In *Proceedings of 4th Working IEEE/IFIP Conference on Software Architecture*, pages 5–14, 2004.

[TOH+99] Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton, and Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21th IEEE International Conference on Requirements Engineering Conference*, Los Angeles, California, USA, 1999.

[UT02] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, 2002.

[vGBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the notation of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 45, 2001.

[WA04] Jon Whitter and João Araújo. Scenario Modeling with Aspects. In *IEEE Proceedings of Software Special Issue*, 2004.

[WLS+07] Hai H. Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff

Pan. Verifying feature models using owl. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5:117–129, 2007.

[YLM04] Yijun Yu, Julio Leite, and John Mylopoulos. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. In *Proceedings of12th IEEE International Conference on Requirements Engineering*, Kyoto, Japan, 2004.

[ZK13] Qinglei Zhang and Ridha Khedri. Proofs of the convergence of the rewriting system for the weaving of aspects in the AO-PFA language. Technical Report CAS-13-01-RK, McMaster University, Hamilton, Ontario, Canada, March 2013. Available: `http://www.cas.mcmaster.ca/cas/0template1.php?601`.

[ZKJ11] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An aspect-oriented language based on product family algebra: Aspects specification and verification. Technical Report CAS-11-08-RK, McMaster University, Hamilton, Ontario, Canada, Nov 2011. Available: `http://www.cas.mcmaster.ca/cas/0template1.php?601`.

[ZKJ12a] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An aspect-oriented language for product family specification. In E. Shakshuki and M. Younas, editors, *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies*, volume 10 of *Procedia Computer Science, ANT 2012 and*

MobiWIS 2012, pages 482 – 489, Niagara Falls, ON, Canada, August 2012.

[ZKJ12b] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. Verification of aspectual composition in feature-modeling. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods, 10th International Conference, SEFM 2012*, volume 7504 of *Lecture Notes of Computer Science*, pages 109–125. Spinger Berline / Heidelberg, Thessaloniki, Greece, October 2012.

[ZKJ13] Qinglei Zhang, Ridha Khedri, and Jason Jaskolka. An aspect-oriented language for feature-modeling. *Accepted May 2, 2013 to Journal of Ambient Intelligence and Humanized Computing*, 2013.

Index