A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

A Study on Implementations of Feature Interaction in Software Product Lines

ソフトウェアプロダクトラインにおける フィーチャの相互作用の実装に関する研究

Fuminobu TAKEYAMA

Academic Advisers:
Osamu WATANABE
Shigeru CHIBA

December 2013,

Abstract

A software product line is a family of similar software products built from common software artifacts. One approach to develop a product line is to decompose software into features and customize the resulting product by selecting a subset of those features. Feature-oriented programming is a paradigm that offers mechanisms for modularizing features. It allows implementing code related to a feature separately into a *feature module*. The key technique for the separation is destructive class extensions, which can extend the definition of existing classes from the outside of those classes. Developers generate products by combining feature modules. Aspect-oriented programming also supports destructive class extensions, and hence it is usable for implementing feature modules.

However, naive combination of feature modules may cause problematic behavior if the features interact with each other. Feature interaction is essential behavior between a specific combination of features. Programmers must implement interaction so that such combinations of feature modules work correctly. A modular approach is to implement interaction separately in *derivatives*, but the number of the derivatives is too large to maintain.

Interaction in aspect-oriented programming is conflict of aspects, which happens when multiple advices extend the same method. Programmers need to specify how conflicting advices are executed. An incomplete approach for advice composition used in the existing languages is to specify precedence order to conflicting aspect to linearize them. This is because some combinations of aspects do not have acceptable order.

To address these problems, this dissertation proposes two language extensions with mechanisms for implementing feature interaction: FeatureGluonJ and Airia. FeatureGluonJ introduces a new module system for features supporting inheritance. Inheritance of feature module enables to implement a generic derivative reusable for multiple combinations of features, which reduces the total number of derivatives.

Airia, a language extension to AspectJ, provides a novel kind of advice called a *resolver* for advice composition. Resolvers implement aspect composition separately from conflicting aspect. They can execute part of the conflicting aspects and merge the result of the execution by using `proceed()` calls extended in Airia.

Acknowledgments

First of all, I would like to express my profound gratitude to my supervisor, Shigeru Chiba. Without his advice and direction, I could not finish this research. I also would like to express my gratitude to Osamu Watanabe, my supervisor for the last two years in Tokyo Institute of Technology. The dissertation committees are organized by Osamu Watanabe, Shigeru Chiba, Hidehiko Masuhara, Ken Wakita, and Kazuyuki Shudo. I appreciate the feedback offered by them.

My special thanks go to my colleagues in Core Software Group. Especially, Michihiro Horie always showed me a way to a Ph.D. degree. Masayuki Ioki, Hidekazu Tadokoro, and Shumpei Akai had spent about 5 years together with me towards the degrees. YungYu Zhuang, Maximilian Scherr, Kazuhiro Ichikawa, Hiroshi Yamaguchi, and the cheerful master students have often encouraged me.

I also would like to offer my special thanks to Atsushi Igarashi (Kyoto University). Discussion with him was invaluable to improve my understanding of virtual classes.

I am grateful to the JST CREST *Development of System Software Technologies for post-Peta Scale High Performance Computing* project and the *CompView* Global COE program. I received financial support as a research assistant in these projects. I am also grateful to Department of Creative Informatics, the University of Tokyo, where my research activities have been conducted for two years.

Finally, I would like to thank my parents for supporting me for a long time.

Fuminobu Takeyama
December 2013

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

As more and more software products get available these days, how to develop the software products efficiently is becoming a fundamental goal of software engineering. Software products are not only applications running on laptops or web servers. Most modern electrical products equip micro processors; they require their own software. Thereby, the scale of such software products have been increased, and hence developing them from scratch is not possible with regard to both the number of the products and the size of them.

Fortunately, all of the products are not unique. Some products have commonality with each other. Thereby, they can be implemented by reusing a part of their source code or other software artifacts. An example of this scenario is found in the 1970's, which is the development of power-plant controlling systems. Although their software is different for each plant depending on its output scale or requirement for an electric power industry, most essential part of the software is common among them. Generating products from a software template allowed developing such family of similar software products.

Another successful family of software products is the Linux kernel. It is a set of customizable operating system kernel modules, widely used from supercomputers to embedded systems. The developers can choose CPU architectures, device drivers, and functionalities such as file systems, for their own kernels. Various kernels are generated from its single source-code tree; the C preprocessor allows compiling the code regions corresponding to the architectures or the functionalities selected for the kernel. The Linux kernel also provides tools for customizing a kernel and a language for defining selectable components and managing their dependencies.

The methodologies for developing a family of similar software products were established recently; such a family is now called a *software product line.* In 1990, Kang et al. proposed *feature-oriented domain analysis* (FODA) [42], where a product family is decomposed into what are called *features.* A feature is a user-visible property of software products. The CPU architectures and the functionalities of the Linux kernel can be considered as features in FODA. To build a product, developers select features used for it so that the selected features will satisfy the purpose of that product. By classifying features as optional features and mandatory features, FODA helps developers understand the possible variations of the product family. Based on the idea of FODA, various approaches for designing, managing, and evolving product lines have been developed.

The research community has proposed several programming paradigms for software product lines to supersede the approaches by preprocessor directives. Feature-oriented programming (FOP) is a language paradigm that aims modularization of features. Since fragments for different features tend to be tangled in a single module in the preprocessor approach, FOP separates the code fragments for a feature into an independent module. We call the independent module a *feature module.* After selecting the features necessary for a product, that product is generated by compiling the corresponding feature modules. AHEAD tool suite [18] provides a feature-oriented programming language supporting a class-like construct called a *refinement.* Refinements allow grouping the code for each feature by *destructive class extensions* [23]. From the outside of an existing class, they can add new fields to that class. Refinements can also replace methods of that class with their methods. As in a method overriding the corresponding method in the super class, the new method of refinements can invoke the original methods. Programmers can implement part of class declaration separately in a refinement if it is specific to a feature. If the programmers use that feature for a product, they compile the refinement together with the class; the AHEAD compiler inserts fields and/or methods of that refinement to the class.

The support of features is not limited to the languages designed for FOP. Aspect-oriented programming (AOP) allows to separate the code related to a feature into independent module, an *aspect.* In a typical AOP language, AspectJ, aspects can insert new fields into existing classes and can separate code fragments related to a feature into a method-like construct, an *advice.* The code written in the advices will be executed when program execution reaches the points where the code fragments were. Other language mechanisms such as virtual classes [56, 30] also support modularization of features.

1.1 Motivating Problems

Although the existing FOP languages achieve modularization by grouping the code related to the same feature, other properties of modularization for FOP, including composability, code reuse, and information hiding, have been an open question [46]. Composability is the ability that allows to combine multiple modules together without modification but with permissive efforts. Composability is an essential property for developing product lines in FOP because it enables to developers to generate a product from a subset of prepared feature modules without modifying any code.

Naive composition of feature modules may cause problematic behaviour if some of the features interact with each other. *Feature interaction* is both positive and negative effects from one feature to another [20]. It happens in a software product when the product uses those features together. Programmers must implement extra code for the combinations of interacting features so that the features can work as expected and it can produce synergy.

A modular approach is to implement such interaction into an independent feature module, called *derivative* [54]. A derivative is compiled together for products including the interacting features A typical kind of interaction is structural interaction; it happens when one feature needs to perform additional behavior when a methods of another feature is executed. If programmers implement the behavior in that method, it will be executed even in a product without the former feature, and if they naively implement it as a destructive extension to that method, they cannot select the former feature without selecting the latter feature since the products without the latter feature do not contain the extended method. They should separate such a destructive extension into a derivative between those features.

Nevertheless, since product developers should be able to build a product just by selecting necessary features, programmers must prepare derivatives for every possible combination of interacting features. The scalability of the number of derivatives for a product line is still under discussion in the research community; the number of derivatives might become too large to maintain.

Another interaction among features is conflicts among aspects. Aspects conflict when their advices extend the same method and they are to be executed at the same time. The composed behavior of the conflicting aspects is determined by the compiler through the process called *aspect composition.* The behavior might be different from programmers' intention even though each of the aspects is correct. This situation is called *aspect interference*, which have been a crucial problem in the AOP community. Programmers sometimes need to specify how the compiler combines conflicting aspects defined in different feature modules. Some AOP languages including AspectJ provide mechanisms for controlling conflicting aspects and allow programmers to implement derivatives for advice composition by using those mechanisms.

An incomplete aspect-composition approach adopted in AspectJ is to linearize conflicting advices naively according to their precedence order. When the execution of the program reaches the join point where the advices conflict, the advice with the highest precedence is executed first. AspectJ provides a special call proceed() for executing another conflicting advice. However, there is not acceptable order among some combinations of aspects. Moreover, each conflicting advice must contain the code for working with the others; they should contain at least proceed() to invoke another aspect. Since programmers must be aware of other aspects and their advices, aspect conflicts may restrict to independently develop the aspects.

The composability of the mechanisms for advice composition is also a problem. If they are composable, the composition code (*i.e.*, the code for aspect composition) for a set of aspects can be obtained just by combining the composition code for its subsets without modification. While mechanisms dedicated to aspect composition such as declare precedence in AspectJ provide composability, there are approaches that are expressive but are not composable. For example, several AOP systems provide meta-programming facilities for manipulating conflicting aspects. In such systems, it is not possible to obtain the composition of a set of aspects from the methods manipulating its subsets. Besides, if the composition mechanisms are not composable, conflicts among such mechanisms is another problem; we need to introduce another mechanism for the new conflict problem.

1.2 Solution by This Dissertation

This dissertation addresses the problems mentioned above on software product line development in feature-oriented programming by designing two languages: FeatureGluonJ and Airia. FeatureGluonJ is a new feature-oriented programming language based on an existing AOP language, GluonJ [23]. FeatureGluonJ offers language mechanisms for feature interactions and reduces the number of derivatives. The other language, Airia, is a language extension to AspectJ and aims to provides expressible and composable mechanisms for advice composition.

FeatureGluonJ

FeatureGluonJ is a FOP language with destructive class extensions called *revisers*, introduced by GluonJ. Revisers can extend other classes in a similar way to refinements. Thereby, GluonJ can be used for feature-oriented programming. FeatureGluonJ extends GluonJ by adding new language constructs specific to feature-oriented programming. One of them is a module system for features. It enables to explicitly group classes and revisers for each feature.

FeatureGluonJ also provides an inheritance mechanism for feature modules. Features often make an *is-a* relation, *i.e.*, generalization of another feature. In FeatureGluonJ, programmers can implement specialized features as sub-feature modules of an abstract one. FeatureGluonJ adopts virtual classes for the inheritance of the modules consisting of multiple classes. Sub-feature modules inherit classes from the super feature module; at that time, the sub-feature modules may override the inherited classes to add the behavior particular to the sub-features. FeatureGluonJ integrates revisers into virtual classes; sub-feature modules also inherit and reuse revisers defined in their super feature module.

The inheritance mechanism facilitates to reduce the number of derivatives. This is because the derivatives among groups of specialized features tend to be redundant. A super feature module works as a programming interface among its sub-feature modules, which specifies virtual classes that the sub-feature modules must contain. FeatureGluonJ provides *generic feature modules*, which are parametric feature modules used for templates of derivatives. A generic feature module takes a feature module from each group as a parameter and implements derivatives for those groups. It accesses feature modules given as parameters via the interfaces their super feature modules define.

Airia

Since aspect composition is not a trivial task, Airia enforces programmers to implement composed behavior of conflicting aspects. In Airia, the composed behavior is implemented in what is called a *resolver* separately from the conflicting aspects. A resolver is a new kind of advice for advice composition, and it is executed only when specified advices conflicts.

Airia still supports linearization to reuse conflicting advices. Moreover, it can execute only a part of conflicting advices and can merge the results of their execution instead of calling one by one by `proceed()` calls written in a resolver. The conflicting advices do not need to be aware of composition.

Airia extends the proceed() call of AspectJ so that it will allow removing unnecessary advices as well as giving precedence order among conflicting advices. The precedence order defined by the proceed() call is effective until the invoked advice returns. A resolver may contain multiple proceed calls that define different precedence order to change it depending on the dynamic context.

Furthermore, resolvers themselves are advices and hence they are also composable by the same language construct, a resolver. If a resolver conflicts at the same join points with other advices, then programmers can describe another resolver that implements the composition between the resolver and the advice. Airia adopts static composition of advices, which is an advantage for finding ambiguity or inconsistency of advice composition at compile time. If resolvers do not define composed behavior consistently, the compiler reports errors.

1.3 Position of This Dissertation

Although a number of programming languages designed for advanced modularity have been proposed, only a few practical product lines are written in such programming languages; most product lines are still written with preprocessor directives even though the modularity is low. One of the reasons is the absence of a mechanism for expressing feature interaction. This dissertation extends the definition of the optional feature problem [48] to cover the problem that the number of derivatives is often huge. The optional feature problem was originally defined as a problem that structural interaction among features cannot be implemented in modules of these features. It has been said that a solution of the problem is separating interactions into derivatives. We believe that the practicality of product-line development in FOP or AOP will be enhanced by this dissertation, which addresses the problem of feature interaction.

The approach in this dissertation follows the history of programming languages. We have introduced new language mechanisms that are expressive but are dedicated to specific problems. A naive approach for addressing problems is using meta-programming. For example, some AOP languages support meta-programming for advice composition. Meta-programming is sufficiently powerful for controlling almost everything in a program. However, in the history of programming languages, implementation techniques using meta-programming have been replaced with different techniques using simpler language mechanisms in order to achieve better understandability or composability. The aspects in AOP are such a simpler mechanism to avoid the use of meta-programming. Airia does not take AspectJ back to the language extensively using meta-programming.

A drawback with this approach is that adding new constructs may make the programming language complex. Our languages have been designed to reduce complexity as much as possible. First, a resolver in Airia is a new mechanism added to AspectJ but it is just a new kind of advice; it is not a completely new mechanism unlike other advice composition mechanism. This decision prevents another composability problem, which requires another mechanism for the composition of that new mechanism. Next, FeatureGluonJ does not support the full capability of virtual classes. Feature modules in FeatureGluonJ are modules only for implementing features and hence they are implicitly instantiated when the features are selected for a product. Since the original aim of virtual classes is to make a reusable library consisting of multiple classes that mutually refer each other, programmers can freely instantiate a module containing virtual classes in the program. In this dissertation, we consider the original semantics of virtual classes is too expressive for FOP.

This dissertation addresses only limited range of feature-oriented product line development. CIDE is an integrated development environment (IDE) that allows modularizing features virtually. Other abstraction mechanisms such as operating systems, frameworks, and shared libraries can achieve variability of software products. For example, operating system allows executing a software product on various devices without modifying the source code of that product.

1.4 The Structure of This Dissertation

Chapter 2: Feature-oriented Software Development

This chapter first explains a development approach of product lines, feature-oriented software development, and programming languages in which features can be modularized. Then, it also mentions feature interactions addressed in this dissertation.

Chapter 3: FeatureGluonJ

This chapter introduces a new feature-oriented programming language named FeatureGluonJ. It provides inheritance mechanism for feature modules. A super feature module works as an interface among its sub-features and enables to implement interaction between multiple combinations of features in a single module to reduce the number of derivatives.

Chapter 4: Airia

For the composition of complicatedly conflicting advices, Airia provides a new kind of advice called a resolver. It can implement composed behavior of conflicting aspects by calling part of them through Airia's extended proceed() call. A resolver can also address conflicts among resolvers and normal advices.

Chapter 5: Concluding Remarks

This chapter concludes this dissertation.

Chapter 2

Feature-oriented Software Development

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems including software product lines [6]. In FOSD, developers concentrate on features consisting the software systems in every stage of the development process, *i.e.*, analyzing application domain, designing, implementing, and testing product lines. A product can be considered as a subset of the features comprising of the product lines.

There are many languages with linguistic mechanisms that can modularize features. One of the mechanisms is destructive class extensions [23] (aka. aspects in aspect-oriented programming). Destructive extensions allow extending behavior and/or attributes of classes from the outside of the extended class. The extensions are not visible from the extended class. This property of the extensions is known as obliviousness [34] in aspect-oriented programming.

The last of this chapter mentions the problems of feature-oriented product lines development addressed in this dissertation. One is the optional feature problem, which is difficulty in implementing structural feature interaction, *i.e.*, a situation in which an optional feature need to extend a class of another optional feature. In product-line development in aspect-oriented programming languages, conflict of aspects at the same join points is another problem. However, the existing mechanisms for aspect composition are not satisfactory because there is no acceptable order among some combinations of advices for naive linearization. Moreover, such composition mechanisms does not provide sufficient composability.

2.1 Software Product Lines

Software product lines (SPLs) refer to families of similar software products. SPLs development facilitates to reduce implementation cost by developing each product by combining what is called *core assets* instead of building it from scratch. The core assets are software artifacts or resources reusable among the product line; developers gain or produce them during the development. First, the software artifacts are categorized as below:

Software components are units of software for abstraction or reuse. They are often represented by language constructs, for example, classes in object-oriented programming languages.

Domain models are abstract representations of a software system from a particular view point. It is often expressed by a graph in the Unified Modeling Language, such as a use case diagram.

Test cases are pairs of inputs and expected outputs against the inputs, which can be source codes for unit testing frameworks.

Documents include both end-user manuals and ones for developers such as API references.

On the other hand, the resources includes knowledge of the problem domain, experience of implementation, and budgets for developing the products.

Figure 1 (1)–(5) depicts a development scenario of a product line. Suppose the development of two similar products, A and B. (1) First, developers break down these products into core assets, which are indicated by rectangles, so that they can reuse a part of the core assets between the products. Next, they develop the core assets. For example, programmers implement classes shared between the products. What can be reused in this scenario is not only the implementations of the classes; since the same programmers involve in implementing both products, it can be considered that they can reuse the experience in one product for the other product. (2) Finally, to build the products, the developers combine the subset of the core assets needed for each of the products.

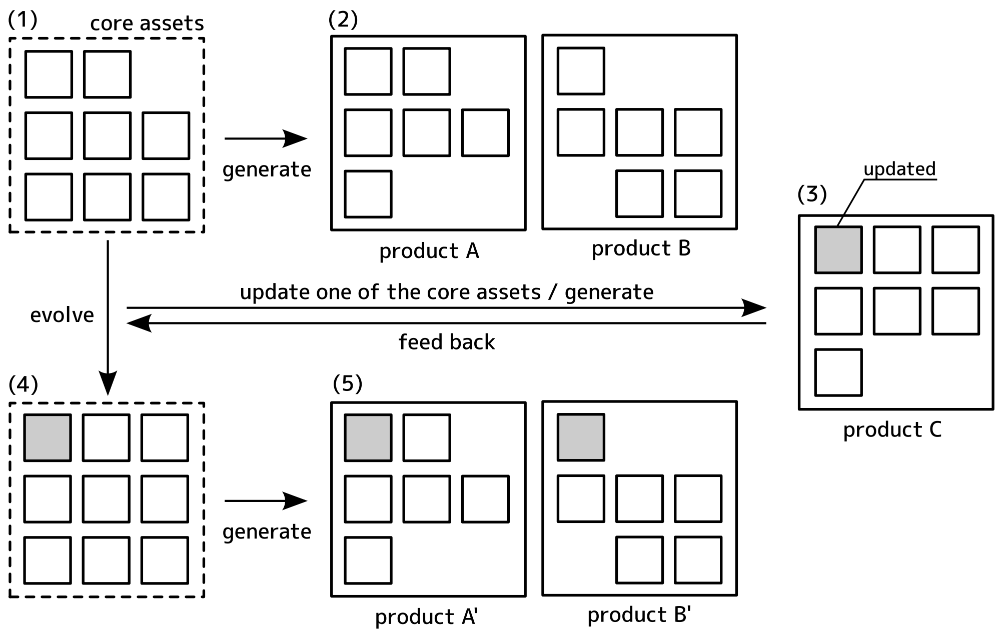

Fig. 2.1: Development of a product line and management of its variations

Evolution of product lines after building initial products is important in product-line development. For the evolution, the developers add new core assets or update the existing ones. (3) Suppose that the company decide to develop a new product, C by adding new functionalities to product A. During the development of C, the developers not only add new core assets but may also modify the core assets that have been already used for A and B. For example, they find security vulnerability in one of the core assets and fix it. They combines these core assets for the product C. (4) The new and updated core assets for C are fed back to the collection of core assets shared among the product line.

Management of variation of product lines is also essential. (5) After the evolution of the product line, the existing variations, A and B, must be still valid; they can be generated without compile errors. Since some core assets used in A and B are updated for C, the resulting products containing the updated core assets are not exactly the same as the original products. Nevertheless, the core assets should be updated so that the existing variations of the product line can provide the same capabilities or functionalities as before while the security problem is resolved.

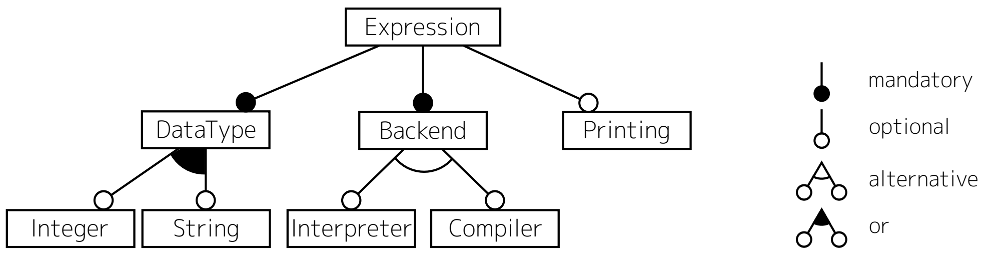

Fig. 2.2: A feature-model diagram

2.2 Features in Software Product Lines

What are criteria for dividing core assets? Kang et al. introduced a new concept, a *feature* and proposed feature-oriented domain analysis (FODA) [42, 43]. They advocate that focusing on features comprising a product line is important for the successful development. This is because a feature is a user-visible characteristic, capability, or functionality of a software system although there are several definitions of a feature. Features can be often found in product development scenario from upper process to lower process. For example, to develop new product C as shown in Section 2.1, the company plans and implements new features for that product. After they complete the product, they advertise the new features for its sales promotion.

In the design phase of a product line, developers analyze features in it and decompose it into features. An important purpose of the decomposition is to understand the commonalities and the variabilities among the products. While the commonalities are features used for all the product in a product line, variabilities are represented by features used only in some products; such features are called *variation points*. Selecting a subset of features generates varieties of products. The variation points must be implemented so as to be unplugged from the other features.

In order to understand features consisting of a product line, feature model is often used for describing the result of feature decomposition. The model is illustrated as a tree diagram. The tree diagram shown in Figure 2.2 is for an expression product line, which provide simple interpreters or compilers for evaluating given expressions. On that diagram, a node represents a feature in the product line, and an edge between the features represents aggregation (aka. *has-a*) or generalization (aka. *is-a*). The aggregation is used when a feature is divided into small features. For example, the relationship between the Expression feature and its children is aggregation. The Expression has three features DataType, Backend, Printing. Product lines often provide specialized variations of features. The generalization is a relationship between such specialized variations and their abstraction. The relationship between the Integer and the String features is generalization; they are generalized by DataType.

Developers can also intuitively understand variations available in the product lines from variation points on that diagram. An edge ending with a white circle indicates an optional feature. The developers may select optional features for products. If they do not select an optional feature, the feature is not implemented in that product. *Alternative* and *or* features are also important variation points. A filled segment (*i.e.*, part of a circle) drawn among children represents *alternative*. Developers choose exactly one from the children. On the other hand, they must at least one feature from *or* features, which are indicated by an arc drawn among children. Note that *alternative* or *or* features are not used for constraints for feature selection such that a feature requires another feature or conflicts with another.

2.3 Automated Generation of Software Products

FOSD aims at automated generation of software products. This is a major difference from other software development styles where developing a product from software-component libraries generically reusable among products. In that development style, programmers need to assemble components implementing features by writing code for initializing and connecting them when they build a product. On the other hand, FOSD allows developers to generate a product just by selecting necessary features from the feature model diagram.

There are several tools that facilitate to select features and build products. They include *guidsl* [15], FeatureIDE [76], Autotools (Autoconf [1] and Automake [2]), and make menuconfig — a tool for customizing a Linux kernel. For example, after developers select features for a product on GUI or CUI to customize their Linux kernel, it defines preprocessor variables assigned to the selected features; features are surrounded by #ifdef–#endif in the source files. The variables are passed to the preprocessor, and only code regions of selected features are compiled together. Programmers do not need to implement additional code for building products. Although tool support is out of the scope of this dissertation, the existing tools can be used for the approaches in this dissertation.

```
class Application {
#ifdef PHOTO
  PhotoListScreen photoListScreen;
  PhotoController photoController;
#endif
#ifdef MUSIC
  MusicListScreen musicListScreen;
  MusicController musicController;
#endif

  public void startApp() {
#ifdef PHOTO
    photoListScreen = new PhotoListScreen();
    photoController = new PhotoController();
#endif
#ifdef MUSIC
    musicListScreen = new MusicListScreen();
    musicController = new MusicController();
#endif
    // : other initialization
  }

  public static void main(String[] args) {
    Application app = new Application();
    app.startApp()
    // :
}}
```

List. 2.1: Implementing an SPL with preprocessor directives

2.4 Approaches for Implementing Features

2.4.1 Conditional Compilation

A conventional approach for generating products by selecting features is making code snippets for unused features removable by conditional compilation. List. 2.1 shows an example of implementation of a product line written in Java with C/C++-like preprocessor directives. In the Application class, code snippets including field declarations, specific to optional features are surrounded by the preprocessor directives.

A drawback with this approach is that the code snippets from different features are tangled in one language construct. In other words, the boundaries between the features are not clear. For example, a feature might accidentally refer to a variable defined by another feature. If programmers change the code referring the variable, it might affect the behavior of other features.

This representation is not suitable for a unit of code for feature-oriented development. Source code is sometimes managed for each source file, for example, in a version control system. A developer responsible for a feature need to take care changes to source files implementing the feature. However the changes may be for the other features sharing the same source files. Moreover, since the source files are reused as core assets among products, a product may have source files that textually contain unnecessary code for that product.

2.4.2 AspectJ

AspectJ [50] is a typical aspect-oriented programming language based on Java. The aim of aspect-oriented programming is to group code that implements the same concern that scatters over a class hierarchy in object-oriented design. Such concerns are called cross-cutting concerns. AspectJ allows to implement cross-cutting concerns into its unique module, *aspect* with its linguistic mechanisms, an *advice* and a *pointcut*. AspectJ achieves the separation by executing code written in advice at appropriate timing of the execution of the program. *Join points* are an abstraction of the timing and the pointcut is a mechanism for selecting join points. Every advice has a pointcut expression to specify the join points when the advice is executed. From the classes under the join points, the concerns separated from the classes into the aspect are not visible. This property is known as obliviousness [34].

A feature can be regarded as a cross-cutting concern; it cannot be implemented by only its classes, but it requires code in classes belonging to other features. Since AspectJ allows separating features into aspects, several product lines such as the feature-oriented version of Berkeley DB [44] and MobileMedia [78] have been developed in AspectJ.

An advice is a method like construct with parameters and a return type, but it is invoked implicitly at the join points specified with its pointcut. AspectJ provides three kinds of advices: `around`, `before`, and `after`. An `around` advice replaces computation at the join points selected by its pointcut. For example, programmers can override existing methods by an `around` advice by using an `execution` pointcut, which picks up join points when the body of the method given to the pointcut is executed. When the method is invoked, the advice is executed as if the method is overridden in a subclass. The advice returns a value instead of the method.

An `around` advice can execute the original method with a `proceed()` call, which is similar to a `super` call in Java. Thereby an `around` advice can wrap the original computations associated with the join points; namely, it adds actions before and after the original computations. The return value of the computation is forwarded to the return value of proceed() call. On the other hand, before and after advices execute additional actions before/after the join points and always run the original action. These advices can be rewritten to around advices. The following before advice:

```
before(): execution(void Application.startApp()) {
  System.out.println("Application started");
}
```

can be rewritten to the next around advice:

```
void around(): execution(void Application.startApp()) {
  System.out.println("Application started");
  proceed();
}
```

A mechanism called an *inter-type declaration* allows an aspect to add a new field and/or method into an existing class. It is written as a field/method declaration in the aspect, but the name of the extended class is specified before the name of the field/method. For example, the following aspect adds a field with the PhotoController type into the Application class:

```
aspect Photo {
  PhotoController Application.photoController;
}
```

Inter-type field declaration can be used to separate field declarations belonging to a feature if the feature is different from the feature of the class containing the fields.

2.4.3 AHEAD Tool Suite

AHEAD [1] Tool Suite [18] provides models, domain-specific languages (DSLs), and tools for step-wise refinements, which enables to develop incrementally large-scale complex programs from a simple program. A goal of AHEAD is to develop SPLs by feature refinements, which is to add features to the cores of the product lines. The origin of AHEAD is in the two customizable families of software products and their generator [17]: Genesis—a customizable database management system—and Avoca—network software suites consisting of communication protocols. These families can be regarded as software product lines.

The contribution of AHEAD is an algebraic model that can represent both code artifacts (*e.g.*, Java source code) and non-code artifacts (*e.g.,* grammar

[1]Algebraic Hierarchical Equations for Application Design

of languages). In AHEAD, a software product is expressed by an equation with the composition operator $\bullet$ and sets of constants. Suppose two features, f and g, in a product line. The features are expressed by the sets of artifacts as the following:

$$\begin{aligned} f &= \{A_f, B_f\} \\ g &= \{B_g, C_g\} \end{aligned}$$

Here, A_f, B_f, B_g, and C_g are artifacts; and A, B, and C are names assigned to the artifacts. Both of those features provide the artifacts named B, which may have different contents. Let us consider these artifacts are classes in Java program in this example. The class B is a class shared by two features.

A product made up with the features f and g is denoted by $f \bullet g$. The composition operator yields a new set of artifacts from both operands. If the operands have the artifacts with the same name, the artifacts are composed recursively by the operator:

$$\begin{aligned} f \bullet g &= \{A_f, B_f\} \bullet \{B_g, C_g\} \\ &= \{A_f, B_f \bullet B_g, C_g\} \end{aligned}$$

Artifacts in the equations might be compound artifacts constituted by other artifacts. AHEAD applies composition operator recursively to such artifacts. Assume that the class B_f contains methods mX_f and mY_f, and B_g contains methods mY_g and mZ_g. $B_f \bullet B_g$ in the equation above results in:

$$\begin{aligned} B_f \bullet B_g &= \{mX_f, mY_f\} \bullet \{mY_g, mZ_g\} \\ &= \{mX_f, mY_f \bullet mY_g, mZ_g\} \end{aligned}$$

Although AHEAD aims composition of general artifacts, it provides a DSL named Jak for describing artifacts that are to be composed into Java code. List. 2.2 shows artifacts in the Jak language. The syntax of Jak language is almost the same as Java. A class-like construct with refines keyword is a *refinement*, which is unique to Jak. Refinements are used for describing artifacts, which are to be composed with classes of other features.

The AHEAD tool suite defines the semantics of the method composition, which are similar to ones of around advices in AspectJ. The method of a refinement placed on the left of the composition operator wraps the right method of a normal class or another refinement. Jak provides Super() instead of proceed() in AspectJ. Let us consider the composition of $Application_{Photo}$ (List. 2.2 (a)) and $Appliction_{Base}$ (List. 2.2 (b)). Both the refinement and the class have the $startApp()$ methods. The resulting Application class is as shown in List. 2.3. Developers can generate a product by giving equation of that product to AHEAD's source-to-source translator.

```
refines class Application {
  PhotoListScreen photoListScreen;
  PhotoController photoController;

  public void startApp() {
    photoListScreen = new PhotoListScreen();
    photoController = new PhotoController();
    Super().startApp();
  }
}
```

(a) *Application*$_{Photo}$

```
class Application {
  public void startApp() }
    // other initialization
  }
}
```

(b) *Appliction*$_{Base}$

List. 2.2: A refinement in the Jak language

2.4.4 GluonJ

GluonJ [22, 23] is an AOP language and a natural enhancement of object-oriented programming language, Java; it provides AOP functionalities without pointcuts and advices, which are typical AOP-specific language mechanisms. GluonJ provides functionalities such as destructive class extensions, limited scope, and modular compilation. They allow implementing features separately and building products from combinations of features.

GluonJ introduces a new language construct called a *reviser* for destructive class extensions. It looks like a subclass of an existing class as shown in List. 2.4. It destructively extends the target class written after the revises keyword from the outside of the class without modifying the source code of the target class. A reviser adds fields and methods defined in it into its target class as inter-type declarations in AspectJ do. If a reviser contains a method that has the name same as one in its target class, the reviser overrides the existing method as an around advice with an execution pointcut does. In List. 2.4, the startApp() method overrides startApp() of Application. The method contains a super call (super.startApp()). The super call corresponds to proceed() in AspectJ and invokes the overridden method.

To implement a product line in GluonJ, programmers implement every feature with revisers and classes that contain only code related to the feature. GluonJ has two stages for code generation. It first compiles those revisers and classes separately regardless of whether or not they are used by a product. Next, it links the classes and revisers. At the link time, developers give only

```
class Application {
  PhotoListScreen photoListScreen;
  PhotoController photoController;

  public void startApp() {
    photoListScreen = new PhotoListScreen();
    photoController = new PhotoController();
    startApp$$one(); //
  }

  public void startApp$$one() {
    // other initialization
  }
}
```

List. 2.3: The resulting class generated by AHEAD

```
class PhotoInitializer revises Application {
  PhotoListScreen photoListScreen;
  PhotoController photoController;

  public void startApp() {
    photoListScreen = new PhotoListScreen();
    photoController = new PhotoController();
    super.startApp();
  }
}
```

List. 2.4: A reviser in GluonJ

the classes and the revisers of the features used for the resulting product. If a reviser is linked to the other part of a product, the reviser extends its target class by adding fields and replacing methods.

Classes and revisers in GluonJ are compiled separately, which is an important capability for developing large-scale software including software product lines. GluonJ achieves separated type-checking with its constructs: requires and using. When a reviser refers fields or methods defined in another reviser X, it must have requires X to ensure that X is always applied before it uses such fields or methods. If a normal class need to access the fields or the methods, programmers must put using X in its source file. Note that GluonJ is mostly modular; it requires global knowledge for reasoning when multiple revisers extends the same class. Approaches such as conditional compilation and AHEAD Jak language does not support separate compilation for each feature; depending on combinations of features, they may cause compile errors since it cannot be ensured that accessed variables or methods are always defined although there are literature on the type system for annotation-based product lines [47].

2.4.5 Prehofer's Feature-oriented Programming

Prehofer is first to introduce a new programming paradigm named feature-oriented programming (FOP) and a language mechanism specialized to features [67]. Although this work did not originate from FODA, he noticed that an object is composed of several features. That language mechanism allows separating code related to a feature from its class into independent a mixin-like module. This separated module is used when an object is instantiated. Programmers select the features used for the object with the extended new expressions.

In Prehofer's FOP, a customizable stack class family is implemented as shown in List. 2.5. It consists of two features, Counter for counting the elements and Lock for preventing other thread from using the stack. The constructs with feature is the module for implementing a feature separately. The Stack feature can be considered as a normal class implementing the basic operations of stacks. IStack and ICounter are the Java interfaces that simply define methods such as push() and inc().

The Counter feature (List. 2.5 (b)) consists of two feature modules, which can be considered as refinements to Stack class. Unlike refinements, the implementation of a feature is divided into two parts; the methods and the fields introduced by the feature; and the methods overriding the existing one. The latter part is especially called a *lifter* and represented by a feature module with a lifts keyword.

For example, to make a stack object with Counter, programmers write:

```
new Counter(Stack);
```

Counter(Stack) represents the class that has fields and methods defined in the Stack feature and the Counter feature, where the methods from Stack are overridden by the lifter of Counter. The Lock feature is implemented in the same way to Counter as shown in List. 2.5 (c). The following code:

```
new Lock(Stack);
```

instantiates a stack object with the Counter feature.

Prehofer's FOP is, however, not sufficient for implementing SPLs since features in SPLs are rather cross-cutting concerns than properties of a single class; code related to a feature spreads over multiple classes in a product. Even though a feature is cleanly implemented in classes, the code that instantiate those classes exists in classes belonging other features including base code, which is necessary for all variations of products and has an entry point of the program.

```
feature Stack implements IStack {
  Node head = null;
  public Object pop() {
    Object result = head.getValue();
    head = head.getNext();
    return result;
  }
  public void push(Object value) {
    Node n = new Node(head, value);
    head = value;
  }
}
```

(a) The Stack feature

```
feature Counter implements ICounter {
  int size;
  private void inc() { size++; }
  private void dec() { size--; }
  public int size() { return size; }
}

feature Counter lifts IStack {
  public Object pop() {
    this.dec(); return super.pop();
  }
  public void push(Object) {
    this.inc(); super.push(o);
  }
}
```

(b) The Counter feature

```
feature Lock implements ILock {
  private ReentrantLock l = new ReentrantLock();
  public void lock() { l.lock() }
  public void unlock() { l.unlock() }
}

feature LockFeature lifts IStack {
  Object Object pop() {
    this.lock();
    Object result = super.pop();
    this.unlock();
    return result;
  }
  void push(Objcet o) {
    this.lock();
    super.push(o);
    this.unlock();
  }
}
```

(c) The Lock feature

List. 2.5: A customizable stack class in Prehofer's feature-oriented programming

2.4.6 Virtual Classes

The programming language BETA introduced virtual classes [56], which are nested classes and overridable attributes of objects as virtual functions (aka. methods) can be overridden by their subclasses. The gbeta language [30] generalized the virtual classes so as to implement concerns involving multiple classes that mutually refers each other. Its goal is similar to aspect-oriented programming. A subclass can add fields to virtual classes of other features and change the behavior of the methods of the classes without modifying them. CaesarJ [12] (including Caesar [61]) and Object Teams [37] support this generalized virtual classes.

Virtual classes have been used to implement feature-oriented product lines. For example, Dungeon SPL [35] is a product line of games that players explore labyrinths collecting useful items. The developer of the SPL adopted virtual classes to separately modularize the features by modifying the classes with virtual classes instead of aspects or refinements. The literature [32] also uses an example of virtual classes implementing features for an interpreter.

CaesarJ

The List. 2.6 shows classes written in CaesarJ and implements a simple interpreter SPL, which supports only integer literal and the addition of integer values. Although CaesarJ is a language based on Java, it uses the cclass keyword to define classes. The Base class contains the code necessary for all variations of the product line. It consists of three nested classes, Expression, Literal, and Plus, representing the AST nodes of the interpreters. Before programmers uses these nested classes, they instantiate the outer class, Base since virtual classes are attributes of an object. The following code instantiates the Literal class:

```
Base base = new Base();
base.Literal lit = base.new Literal();
```

Programmers need to specify an object of the outer class when they access to the nested classes from the outside of the nesting class. Here, base.**new** Literal() denotes instantiation of a class bound to the virtual class reference Literal of base.

In CaesarJ, all nested cclasses are virtual classes. The important capability of virtual classes is that a virtual class can override another virtual class contained by the super class of the outer class. Virtual-class overriding is similar to virtual function overriding in C++ or method overriding in Java. A reference to virtual class is not resolved lexically. As a virtual function call

```
public abstract cclass Base {
  public abstract cclass Expression {}

  public cclass Literal extends Expression {
    protected int i;

    public Literal(int i) {
      this.i = i;
    }
  }

  public cclass Plus extends Expression {
    protected Expression left;
    protected Expression right;

    public Plus(Expression left, Expression right) {
      this.left = left;
      this.right = right;
    }
  }

  public class Application {
    public Expression testTree;

    public abstract void test() {
      this.testTree = this.new Plus(
        this.new Literal(1),
        this.new Literal(2));
    }
  }
}
```

List. 2.6: The base classes for a simple interpreter in CaesarJ

may executes a method defined in the subclass by dynamic dispatch, the reference is resolved dynamically depending on an object of the class enclosing them.

If a subclass provides virtual classes with the same name as ones of its super class, the subclass overrides such virtual classes. List. 2.7 shows the EvaluationFeature class, which is a subclass of Base and contains virtual classes named Expression, Literal, and Plus. Those classes override the corresponding classes of the Base class. Suppose that the actual class of base is EvaluationFeature. Within base, the virtual class reference Expression written in Base refers to Expression defined in EavluationFeature. A virtual class reference from the outside of Base is also resolved depending on the actual class of the outer class. Thus, base.Expression also refers to EvaluationFeature's Expression. Note that a virtual class inherits the methods and fields defined in overridden class; for example, the left and right fields are also available in

```
public cclass EvaluationFeature extends Base {
  public abstract cclass Expression {
    public abstract int eval();
  }

  public cclass Literal {
    public int eval() {
      return i;
    }
  }

  public cclass Plus {
    public void eval() {
      return left.eval() + right.eval();
    }
  }

  public cclass Application {
    public void test() {
      super.test();
      System.out.println(testTree.eval());
    }
  }
}
```

List. 2.7: The evaluation feature implement as a subclass of Base

EvaluationFeature's Expression.

A subclass of Base can modularize a feature. The EvaluationFeature class adds the interpreter the capability for evaluating expressions crosscutting over AST classes as shown in List. 2.7. It adds eval() methods for every class; the method returns the result of the evaluation. A product is represented by an object of EvaluationFeature:

```
class InterpreterWithEvaluation {
  public static void main(String[] args) {
    EvaluationFeature product = new EvaluationFeature();
    product.test();
  }
}
```

This product always uses AST classes extended by EvaluationFeature. To make an expression for testing this interpreter, it executes the buildTestTree() method defined in List. 2.6 on the product variable through the test() method. The buildTestTree() method instantiates the Plus class assigned to this. Since actual class of this is EvaluationFeature, Plus refers to Plus of EvaluationFeature, not Base.

CaesarJ supports to build a product from multiple features. List. 2.8 shows another feature that allows the interpreters to print expressions as

```
public cclass PrinterFeature extends Base {
  public abstract cclass Expression {
    public abstract String print();
  }

  public cclass Literal {
    public String print() {
      return Integer.toString(i);
    }
  }

  public cclass Plus {
    public String print() {
      return left.print() + " + " + right.print();
    }
  }

  public cclass Application {
    public void test() {
      super.text();
      System.out.println(testTree.print);
    }
  }
}
```

List. 2.8: The printer feature for the expression product line

character strings. It is also implemented as a subclass of Base; it extends AST classes defined in the super class and add print() method to each of them.

Since CaesarJ supports multiple inheritance of classes, a product with the evaluation and the printer features is represented by a class inheriting from EvaluationFeature and PrintFeature. The class can be implemented as shown in List. 2.9. CaesarJ uses & operator in order to specify multiple classes to an extends clause. Note that CaesarJ adopt mixin composition, a style of multiple inheritance. If multiple super classes contains virtual classes with the same name, they merge the virtual classes into one virtual class. For example, the ProductWithEvalPrint class inherits the Plus classes from its super classes and their super common super class also contains the Plus class. In this case, PrintFeature's Plus overrides one of Base, and then EvaluationFeature's Plus overrides one of PrintFeature. Thus, within the ProductWithEvalPrint, both the eval() and the print() methods are available.

```
public cclass ProductWithEvalPrint
      extends EvaluationFeature & PrinterFeature {

  public void test() {
    Expression expr = this.buildTestTree();
    System.out.println(expr.print() + " = " + expr.eval());
  }

  public static void main(String[] args) {
    ProductWithEvalPrint product = new ProductWithEvalPrint();
    product.test();
  }
}
```

List. 2.9: The evaluation feature written as a subclass

2.4.7 Hierarchical Implementations of Features

Object Teams and CaesarJ adopt another implementation architecture of features, which is hierarchical implementations of features, to implement the authentication and authorization library product line [39], and a stock information broker application [62], respectively. Note that these languages aim at dynamic selection of features differently from languages for static composition of product lines. They support to implement a relation among objects as a feature. They provide reusability of classes among different product lines by adapting classes of a feature to other classes manually and flexibly.

The characteristic of those languages is that they provide both destructive extensions and virtual classes, which is considered as a non-destructive mechanism. CaesarJ allows extending the behavior of arbitrary classes by an AspectJ-like advice although virtual class can extend the behavior of another class nested by its super class. On the other hand, Object Teams can forward a method call on an object to another method defined in what is called a *role class*. This mechanism is called *call-in binding*, and it can replace methods of any classes with ones of role classes.

Those languages introduce AspectJ-like advices and call-in binding for the different purpose from that of virtual class overriding. In an object-oriented implementation of features, a feature is often not behavior of a single object but behavior of a collaboration of classes [39]. Such a collaboration of classes fits to a set of virtual classes nested by the same class, and the outer class represents the feature.

Inheritance of the outer class and virtual class overriding is used to implement features specializing an abstract feature. Those features are illustrated

```
public class Application {
  public void start() {
    //
  }

  public static void main() {
    Application app = new Application();
    app.start();
  }
}
```

List. 2.10: The common part of the interpreter product lines

as alternative features or *o*r features on the feature model diagram. A class and its virtual classes implement the common code among the features, and its subclasses implement them by overriding the virtual classes.

Every feature must contain code extending behavior of classes of other features. Otherwise, the code of the feature will be never executed. Hierarchical feature implementation uses destructive extensions for this purpose instead of virtual class overriding. The implementation design only with virtual classes, mentioned in Section 2.4.6, requires that all the classes extended by some features should be nested classes of the Application class. However, since destructive extensions allow extending classes whenever the classes are defined, programmers only need to separate classes and destructive extensions implementing the same feature into a class in hierarchical designs.

The remaining part of this section explains the syntax of CaesarJ and Object Teams for hierarchical designs. As List. 2.10 shows, the Application class is now an independent class shared by multiple features, which will be used for the following two examples written in those languages.

In CaesarJ, the features that provide operations of expressions are represented by subclasses of ExpressionBase in List. 2.11. It contains an around advice, which is executed when the Application.start() method is executed, and splits code of the feature from the other part. A constructor-call-like constructs without new is unique to CaeasrJ and returns a wrapper object of the Application class. CaesarJ uses wrapper objects to keep values related to features without providing inter-type declarations in AspectJ. CaesarJ facilitates to implement a wrapper class with its wraps clause. The wrapper object is an instance of the ApplicationWithTest class, which is a virtual class of ExpressionBase. It keeps an expression used in the test() method. The wrapper accesses always returns the same object for the same triple of the value before the dot, the wrapper class, and wrapped object.

In List. 2.12, a subclass of ExpressionBase, Evaluation implements the Eval-

```
public abstract cclass ExpressionBase {
  public abstract cclass Expression {}

  public cclass Literal extends Expression {
    protected int i;
    public Literal(int i) {
      this.i = i;
    }
  }

  public cclass Plus extends Expression {
    protected Expression left;
    protected Expression right;
    public Plus(Expression left, Expression right) {
      this.left = left;
      this.right = right;
    }
  }

  public cclass ApplicationWithTest wraps Application {
    protected Expression testTree;

    public void test() {
      testTree = new Plus(new Literal(1), new Literal(2));
    }
  }

  void around(Application t):
    execution(void Application.start()) && this(t)
  {
    proceed(t);
    this.ApplicationWithTest(t).test();
  }
}
```

List. 2.11: The super class for the features providing operations of expressions (CaesarJ)

```
public deployed cclass Evaluation extends ExpressionBase {
  public abstract cclass Expression {
    abstract public int eval();
  }

  public cclass Literal {
    public int eval() {
      return i;
    }
  }

  public cclass ApplicationWithTest {
    public void test() {
      super.test();
      System.out.println(testTree.eval());
    }
  }

  public cclass Plus {
    public void eval() {
      return left.eval() + right.eval();
    }
  }
}
```

List. 2.12: The Evaluation feature in the hierarchical designe (CaesarJ)

uation feature, which provides the evaluation of the expressions. A subclass inherits advices from its super class. The Evaluation class has an around advice defined in ExpressionBase. If this in the advice refers to an Evaluation object, the wrapper access returns an object of the ApplicationWithTest redefined in Evaluation since a subclass also overrides wrapper class in the same way as normal virtual classes.

Although advices in AspectJ are implicitly called if they are compiled, advices in CaesarJ are not executed until programmers manually *deploy* the objects containing the advices. The Evaluation class has the deployed modifier. It indicates that the class is implicitly instantiated, and the around advice is executed when the program execution reaches its join points. If the Evaluation class is compiled, its advice calls the test() method of Evaluation's ApplicationWithTest. In CaesarJ, programmers instantiate an object with advices manually and deploy it by using a deploy statement. An advice can be executed multiple times if programmers deploy multiple objects of the class containing it.

In ObjectTeams, a feature is implemented as a *team class* consisting of virtual classes, called *role classes*. It supports to implement a team by inheriting another team as CaesarJ does. List. 2.13 shows another implementation of the ExpressionBasis, which was shown in List. 2.11. The ExpressionBasis class contains four role classes although the Test class is not a nested class of ExpressionBasis. import team declaration enables to write role classes of a team in an independent source file separately from the team definition.

To extend the behavior of class in the outside of the team, Object Teams provide *r*ole binding, which binds a role instance to an instance of the class specified in a playedBy clause. The Test role of the ExpressionBasis is instantiated implicitly and bound to an object of Application similarly to the wrapper class in CaesarJ. Object Teams also provide *c*all-in binding, which allows forwarding method calls on classes playing a role to the method of the role. It is similar to advices in AspectJ or CaesarJ. The <– operator and the replace keyword written in the last of the Test role are constructs for *c*all-in binding. The replace keyword replaces calls to the method on the right hand side with a method on the left hand side as an around advice does. By test <– **replace** start, calling the start method on an Application object invokes the test method of the Test class.

The Evaluation feature in List. 2.14 is represented by a sub-team of EvaluationBase in Object Teams as well as CaesarJ. The language semantics related to sub-team is similar to CaesarJ; sub-teams inherit all the roles from their super team. The Evaluation team overrides the Test role class to run evaluation of an expression. The Test role of Evaluation inherits a call-in binding and is played by Application. Note that Object Teams require to activate call-in binding manually by the activate method of each team. Otherwise, the test() method is never executed. In this example, only the Evaluation team is activated as shown in the bottom of List. 2.14. Thus, only the Test role overridden by Evaluation is executed.

2.4.8 Delta-oriented Programming

Delta-oriented programming (DOP) [70] is a language paradigm designed for developing software product lines. DOP introduces a new concept, program deltas and allows implementing a feature as a set of program deltas. A delta is comprised of addition and deletion of program fragments. A delta can add new methods implementing a feature into an existing class as refinements of AHEAD can. Moreover, it can remove unnecessary classes or methods. DOP, thereby, enables the product lines style in which developers implement a product with full features and implement another feature by removing unnecessary code from that product.

List. 2.15 is the implementation of the Evaluation feature written in DeltaJava, a DOP extension to Java. DeltaJava provides a language construct,

```
public team class ExpressionBasis {
  public abstract class Expression {}

  public class Literal extends Expression {
    int i;
    public Literal(int i) {
      this.i = i;
    }
  }

  public class Plus extends Expression {
    protected Expression left;
    protected Expression right;

    public Plus(Expression left, Expression right) {
      this.left = left;
      this.right = right;
    }
  }
}
.............................................................................
import team ExpressionBase;

public class Test playedBy Application {
  protected Expression testTree;

  callin void test() {
    base.test();
    testTree = new Plus(new Literal(1), new Literal(2));
  }

  test <- replace start;
}
```

List. 2.13: The super class for the variations of the feature (Object Teams)

```
public team class Evaluation extends ExpressionBase {
  public abstract class Expression {
    public abstract int eval();
  }

  public class Plus {
    public int eval() {
      return left.eval() + right.eval();
    }
  }

  public class Literal {
    public int eval() {
      return i;
    }
  }

  public class Test {
    callin void test() {
      tsuper.test();
      System.out.println(testTree.eval());
    }
  }
}
.................................................................................
Evaluation e = new Evaluation();
e.activate();
```

List. 2.14: The Evaluation feature in the hierarchical designe (ObjectTeams)

```
delta DEvalationFeature when Evaluation {
  modifies class Expression {
    adds int eval() {}
  }

  modifies class Literal {
    adds int eval() {
      return i;
    }
  }

  modifies class Plus {
    adds void eval() {
      return left.eval() + right.eval();
    }
  }
}
```

List. 2.15: The Evaluation feature in DeltaJava

delta modules, representing program deltas. It consists of additions, modifications, and removals of classes. In the delta module for the Evaluation feature, class definitions with modifies keyword are class modifications, which correspond to refinements of AHEAD and extend the existing versions of those classes.

In DOP, multiple delta modules may implement a feature. A when keyword of a delta module specifies the condition on which the module takes effect. Every feature has a variable that become true if and only if the feature is selected, and that condition is represented by logical expression consisting of such variables. The DEvaluationFeature in List. 2.15 has when Evaluation, and hence it modifies classes when the Evaluation feature is selected.

2.4.9 The Package Templates

The package templates [14, 52] provides mechanisms for reusing a set of classes based on virtual classes. A *template* is a reusable module consisting of virtual classes. If a template is instantiated in a *package*, the virtual classes in the template become available in the package, and such virtual classes are overridable. A template can be also instantiated within other templates to define a new template by reusing that template. While instantiating a template, the package templates allows to rename virtual classes defined in the template. As CaesarJ supports, it also allows to merge virtual classes into a virtual class.

List. 2.16 shows the common classes of the interpreter product line written in the package templates. List. 2.17 implements the Evaluation feature. To reuse the virtual classes defined in the ExpressionBase template, The Evaluation template instantiates the ExpressionBase template by the inst declaration, and then overrides the classes contained in the instantiated template by adding methods and/or fields, which are given after the adds keywords.

A product is represented by a *package* and its nested classes. The Program package in List. 2.18 implements a product with the evaluation feature. It instantiates the two templates, Evaluation and Base. It renames classes defined in the Evaluation template. The Exp class is available as the Expression class in the Program package. Two Application classes are merged to one Application class in the package. One of them is defined in Evaluation and renamed from App to Application. The other Application class is defined in Base template. Although both Application classes have the start() methods, package templates can not linearize these methods. Programmers must implement a new start() in the Program package and call those methods manually by using Super[...] calls, which invoke the methods of given merged classes.

```
template ExpressionBase {
  abstract class Exp {}

  class Lit extends Exp {
    int i;
    Literal(int i) {
      this.i = i;
    }
  }

  class Add extends Exp {
    Expr left;
    Expr right;

    Add(Exp left, Expr right) {
      this.left = left;
      this.right = right;
    }
  }

  class App {
    void startApp() {
      test();
    }

    Expr testTree;

    void test() {
      this.testTree = this.new Plus(
        this.new Literal(1),
        this.new Literal(2));
    }
  }
}
.................................................................................
template Base {
  class Application {
    void startApp() {
      // initialization
    }
  }
}
```

List. 2.16: The base classes of the expression product line in the package templates

```
template Evaluation {
  inst ExpressionBase;

  class Expression adds {
    int eval();
  }

  class Literal adds {
    int eval() {
      return i;
    }
  }

  class Plus adds {
    int eval() {
      return left.eval() + right.eval();
    }
  }

  class Application adds {
    void test() {
      super.test();
      System.out.println(testTree.eval());
    }
  }
}
```

List. 2.17: The evaluation feature in the package templates

```
package Program {
  inst Evaluation with
    Exp => Expression,
    Lit => Literal,
    Add => Plus,
    App => Application;

  inst Base;

  class Application adds {
    void start() {
      Super[Application].start();
      Super[App].start();
    }
  }
}
```

List. 2.18: The package representing a product with the evaluation feature

2.5 Implementations of Feature Interaction

When we develop products by combining features, the features used together sometimes interact with each other. *Feature interaction*, coined in the telecommunication system community, is both positive and negative effects from one feature to another [20]. Especially the negative effects are called *interference*. Developers need to carefully treat combinations of features so that they can obtain positive effect from the features and avoid interference. The typical example of the feature interaction is interaction between a call-waiting feature, which answers automatically to ask the caller to wait for a moment, and a call-forwarding feature, which forwards call to another phone. When these features are used together, the useful interaction between the features is to concurrently forward a call to another phone while answering to the caller.

However, the existing feature-oriented programming approaches do not provide sufficient support to deal with the interaction. Thus the composability, an ability to use multiple modules together without modifying them, for building product lines is limited. This dissertation addresses the problem on a mechanism for implementing extra code that makes interaction good effects; the problem is called the optional feature problem.

The aspect-oriented programming community also has been studying interaction of aspects [69]. Especially, several works aims at detecting and/or resolving interference caused by independent aspects woven into the same join points [3, 25, 26, 27, 49, 53, 57, 58, 63, 74]. Such join points are called *shared join points* [63]. This dissertation describes such a situation as an advice conflict at shared join points. The advice conflict is a significant factor of the interference. When the execution of a program reaches at a join point shared by multiple advices (aspects), it is not trivial how these advices are executed. Moreover, since an advice may changes the contexts, the other conflicting advices at the join points might be executed under the unexpected environment. While some languages including AspectJ supports `around` advices, it may prevent the other conflicting advices to be executed.

Typical AOP languages linearize conflicting advices according to the precedence order. An approach for avoiding the interference is to allow programmers to control that precedence order so that the conflicting advices behave as expected. However, among some combinations of advices, there is no correct precedence order with which the composed behavior of the advices is acceptable.

2.5.1 The Optional Feature Problem

Naive combination of features does not produce synergies between them. Developers need to implement code to obtain the useful interaction between the features. The features of the telecommunication system require extra code that checks whether both of the call-waiting feature and the call-forwarding feature are enabled and executes them in parallel.

There is another type of interaction, called structural interaction, which is found in the customizable stack family implemented in Prehofer's feature-oriented programming in List. 2.5. The Counter feature and the Lock feature interact because the size() method introduced by Counter must lock the stack by using the methods of Lock when a product uses those features together. If this interaction was implemented straightforwardly, the size() method would be:

```
public void size() {
  this.lock();          // Lock
  int result = size;    // Counter
  this.unlock();        // Lock
}
```

Since code implementing the interaction is necessary only when the combination of features used together, the code above cannot be implemented in either of the interacting features if the features are optional. The Counter feature cannot contain that size() method since it causes unexpected dependency; the Counter feature cannot be compiled without the Counter feature.

A modular approach is to implement such interaction separately in a lifter or a *derivative* [54]. The interaction between Lock and Counter can be separated as the next lifter:

```
feature Lock lifts Counter {
  public void size() {
    this.lock();
    super.size();  // executes the Counter feature
    this.unlock();
  }
}
```

This size() method overrides the definition of size() in Counter; the super.size() executes the original method in Counter. This lifter is automatically merged into a stack class only when both Lock and Counter is used as the following:

```
new Lock(Counter(Stack));
```

The derivative, which is almost the same to a lifter, is a module that contains only refinements to methods introduced by another feature. It is represented as a partial derivative in AHEAD's algebraic model. The derivative module between Counter and Lock is written as ∂counter/∂Lock where *counter* indicates introductions of new methods and fields by the Counter feature.

A derivative is now used as a terminology for a module implementing interaction. It may also include fields definitions used for the interaction, not only methods overriding to existing ones. The capabilities necessary for implementing a derivative are the same as ones for implementing features. Thereby, language mechanisms such as aspects are sufficient for derivatives. A derivative can be considered as a special feature module that is selected only when the specific combination of features is used together.

The scalability of derivatives is, however, still under discussion in the research community. Suppose that n optional features interact with each other. There are $2^n - n - 1$ combinations of features among the n features, and every combination requires its own derivative in the worst case.

If the derivatives are composable, then they facilitate to reduce the number of derivatives. Prehofer's FOP advocates that if developers prepare lifters for every *pair* of interacting features, then the lifters can implement interaction between any combinations of the features. For example, if there are a lifter between A and B and one between C and D, interaction among A, B, C, and D can be obtained just by combining these lifters into products. Although it is not clear that this statement is true for all cases, the number of the necessary lifters thereby can be reduced to $\frac{1}{2}(n^2 - n)$.

In practical product lines, all the features do not interact with each other, but they still require a number of derivatives. For example, in Berkeley DB refactored in FOP [44] consisting of 38 features, 53 pairs of features have implementation dependencies, which must be separated into derivatives [48]. The paper [46] concludes that the difficulty in implementing features is mainly due to the interaction among the features.

In this dissertation, the optional feature problem refers to the difficulty in implementing the feature interaction. One is that there is no suitable feature for implementing interaction on a feature model diagram; the feature interaction is sometimes an implementation issue [48], and hence the interaction is not aware during analyzing the features. Another is unmaintainability due to a huge number of derivatives. The effort to implement the derivatives will increase as the size of the product line grows up.

Delta-oriented programming is designed so as to address the optional feature problem. A when clause of a delta module may have not only a single variable representing a feature but also expressions constructed from the variables and logical operators. Thereby a delta module that is applied only when Count and Lock are selected is implemented as the following:

```
delta DCountLock when Count && Lock {
  modifies class Stack {
    ...
  }
}
```

Nevertheless, delta-oriented programming does not provide mechanisms to reduce the number of such delta modules. We believe that it cannot solve the entire optional feature problem in this dissertation.

2.5.2 Aspect Interference Caused by Conflicting Advices

2.5.2.1 Aspect Composition

When multiple advices (aspects) conflict at join points, compilers or language runtimes must determine how they execute those advices. This process is called aspect composition [63].

The aspect composition is not often a trivial task. Programmers may need to give hints for the aspect composition to the compiler or explicitly specify code that defines how conflicting advices should be executed. Otherwise, those advices may cause interference. A naive composition of conflicting advices is to execute them sequentially; at the shared join point, one of them is firstly executed, and then the other is executed. Programmers need to carefully give the execution order of the advices since different order sometimes leads a program into different results [3].

The initial work on aspect interaction was done by R. Douence et al. They developed a formal framework to detect conflicting aspects [27, 26]. It also allows programmers to specify composition code explicitly. The framework introduces a unique model of aspect called *stateful aspects*. A stateful aspect is a sequence of pairs consisting of a crosscut (*i.e.* pointcut) and an insert (*i.e.* advice body). Their semantics and representation are based on communicating sequential processes. An aspect is written in the following form:

$$A_{click} = \mu a.\ \hat{}\mathsf{onClick()} \triangleright \mathsf{skip};\ \hat{}\mathsf{onClick()} \triangleright \mathsf{print(''Clicked\ twice!'')};\, a$$

where the left of $\triangleright$ is a crosscut, and the right is an insert. Every aspect has state indicating the current pairs of a crosscut and an insert. The initial state is the first pair of the aspect. Whenever the program execution advances, the framework evaluates the current crosscut; if the crosscut matches to the current join point, then it executes the current insertion and shifts the current state to the next pair. Since skip means doing nothing, the aspect above prints the message when the onClick() method is called twice. Note that

$\mu a.\ A_1; \ldots; A_n; a$ represents that the last a is replaced with the expression following $\mu a.$ recursively.

When a program has multiple aspects, the aspects are combined with parallel $||$ operators. Those aspects might conflict with each other. Suppose that two aspects:

$$A_{click} \ || \ A_{log}$$

are combined. The framework checks if each of the current crosscuts of A_{click} and A_{log} matches to the current join point. If both crosscuts matches to the same join point, it executes $(I_{click} \bowtie I_{log})$ where I_1 and I_2 are current inserts of A_{click} and A_{log}, respectively. The $\bowtie$ is a non-deterministic function representing a conflict, which returns $(I_{click};\ I_{log})$ or $(I_{log};\ I_{click})$. For example, $A_{click} \ || \ A_{log}$ conflicts if:

$$A_{log} = \mu a.\ \hat{}\mathsf{onClick()} \triangleright \mathsf{print("onClock()")};$$

However one of $(I_{click};\ I_{log})$ or $(I_{log};\ I_{click})$ might not be acceptable. To avoid the interference or the ambiguity of the resulting behavior of the conflicting aspects, the framework allows programmers to specify transformation of the $\bowtie$ operators in order to define how $(I_{click} \bowtie I_{log})$ is to be executed. The transformation is defined separately from the aspects. For example, the following transformation:

$$A_{click} \ ||_{seq} \ A_{log} \quad where\ seq = \mu a.\ \mathsf{true} \triangleright (id \oplus seq);\ a$$

always transforms $(I_{click} \bowtie I_{log})$ into $(I_{click};\ I_{log})$ during the evaluation of those aspects. Since this *seq* transformation is often used, the framework provides it as one of generic transformations, $||_{\overline{seq}}$. The framework can check if the aspects are transformed explicitly and have no conflict.

2.5.2.2 Limitations of Naive Linearization

A typical approach for advice composition is linearization of conflicting advices. AspectJ adopts this approach. Assume that there are two around advices woven at the same join point. The AspectJ compiler implicitly combines them. When the program execution reaches at the join point, one of the conflicting advices is executed. If those advices are not defined in the same aspect, it is undefined which advice is selected. Then if the first advice call proceed(), it invokes the other advice. When the second advice returns a value, it becomes the return value of the proceed() call, and the execution goes back to the first advice. The second advice may execute original computation at the join point by its proceed() call.

If the implicit composition causes interference, programmers need to combine them manually. The only mechanism for advice composition in AspectJ is to change the precedence order among aspects. At the shared join point, the advice of the aspect with the highest precedence is executed first, and its proceed() call invokes the advice with the second highest one. In AspectJ, programmers can specify the precedence order as the following:

```
declare precedence: A1, A2;
```

Here, A1 and A2 are names of aspects. This declares that aspect A1 has higher precedence than aspect A2.

However, naively changing the order of advice execution is not sufficient to resolve interference among conflicting advices. Some combinations of aspects have no acceptable order. AspectJ does not provide mechanisms by which programmers add extra computation so that such conflicting advices work in a coordinated manner. The limitation of the linearization is common to the method composition of mixins, which is incomplete classes with implicit super classes for multiple inheritance. Suppose that several mixins have different definitions of methods with the same name. If a class inherits those mixins, those methods are linearized in that class and executed one by one with their super() calls. A suitable ordering of such methods may be difficult to find or may not exist [28, 71].

A workaround for executing such aspects/methods in the linearization is to modify every advice/method by taking others into account. However, it causes another problem, dispersal of *glue code* [28, 71]. The glue code is code necessary for the composition with another conflicting advices/methods executed next. At least, to execute code written in all the conflicting advices/methods, each of them needs to call super()/proceed() and glue the results of that call and the computation unique to itself. This workaround is not possible if the aspects are implemented independently; the programmers of one aspect cannot be aware of the others. If the aspects are parts of libraries, they cannot modify those aspects at all.

Since the naive linearization is not sufficient, several AOP systems provide mechanisms for preventing a part of conflicting advices from being executed. The stateful aspect framework allows programmers to define a new transformation executing only one of the conflicting advices. The following transformation executes only A_{click} when the aspects conflicts after the quiet() method is executed:

$$A_{click} \mid\mid_f A_{log} \quad where\ f = \hat{}\mathsf{quiet()} \triangleright (id \oplus \bowtie);\ \mu a.\ \mathsf{true} \triangleright (id \oplus fst);\ a$$

The $(id \oplus fst)$ transforms $(I_{click} \bowtie I_{log})$ to (I_{click}). Note that the composition is also stateful. Programmers define transformations in the same way as aspects and hence they can transform $\bowtie$ to the composed behavior according to the dynamic context.

Reflex AOP kernel is a library that introduces reflective APIs for AOP to Java. It provides powerful operators for aspect composition. A characteristic ability of the operators is to execute two around advices sequentially without linearizing them. Suppose conflicting advices r1 and r2. By using Reflex's ord operator, if ord(r1, r2) is given to the kernel, r1 is executed first and proceed() calls in r1 invokes original computation at the join point instead of r2. After r1 exits, r2 is executed, and it may executes the original computation by proceed. A composition operator for linearization is also available in Reflex. To linearize r1 and r2, programmers gives nest(r1, r2) to the kernel.

With regard to method composition in object-oriented programming, traits [28, 71] resolve the dispersal of glue code of mixin, enforcing programmers to write a glue method when different traits contain methods with the same name. In the glue method, programmers must invoke the conflicting method directly and explicitly merges the result of the conflicting methods. Suppose a class built from two traits with toString() methods. The methods defined in those traits become available as if they are defined directly in that class due to the *flattening property*. No method in the traits takes precedence over the method of the other trait. To avoid the method conflicts, a composed class can rename methods imported from the traits. Programmers need to change the toString() methods to toString1() and toString2(), for example. Otherwise, that class is invalid and results in a compile error. They can implement a new toString() method in that class; it may call toString1() and toString2() and concatenate the results of those methods.

Multi-dimensional separation of concern [75], subject-oriented programming [24, 65], and their implementation, Hyper/J offers mechanisms for merging methods from different modules implementing a subject (*i.e.*, a feature) into one. Programmers must explicitly how conflicting advices are executed. A mechanism unique to Hyper/J is a *summary function*. It is a function that receives the return values of conflicting methods as an array and reduces them to one value. Programmers can write a summary functions in Java code.

These expressive composition mechanisms, nevertheless, cannot be naively applied to AOP languages such as AspectJ. One difficulty is support for around advices with return values. Stateful aspects do not have around advices, and it is not clear how Reflex treats the return values of around advices executed sequentially.

Another difficulty is that conflicts of advices are fluid compared with traits or modules implementing subjects. Although a class combining traits can reason which methods conflicts statically, advices may be executed on the one join point but may not be executed on another if they have a pointcut depending on dynamic context. Thereby, it is difficult to invoke conflicting advice directly.

2.5.2.3 Composability of Composition Code

Composability of mechanisms for advice composition is also an important property of AOP language. If they are composable, the composition code (*i.e.*, code for aspect composition) for a set of aspects can be obtained just by combining the composition code for its subsets without modification. Suppose that there are two sets of aspects {A, B} and {C, D}, and each set has composition code. For example, in AspectJ, the composition code can be written as the following:

```
aspect AandB {
  declare precedence: A, B, X;
}

aspect CandD {
  declare precedence: X, C, D;

  void around(): ... {
    ...
  }
}
```

where X is a commonly used aspect. When programmers need those five aspects together in an application, they can reuse those aspects implementing composition for those five aspects since declare precedence is partially composable. They simply need to compile AandB and CandD together for that application. The composed composition code is equivalent with:

```
aspect AandBandC {
  declare precedence: A, B, X, C, D;
}
```

AspectJ and other AOP systems adopt a declarative approach using precedence for the aspect composition. A benefit of this approach is stability with regard to whether aspects are actually executed or not. Even though the aspect B is not woven at the join point, the composition code above is still valid. When aspect A calls proceed(), it simply invokes an advice of X instead of B. If the advice executed next was explicitly defined as B::m() as in C++, it would be fragile with regard to the changes of the linearized advices (methods) [71, 28].

Another benefit is that programmers can insert new aspects into an existing execution order. Assume that new aspect Y, which should be executed between A and B. AspectJ allows such composition by declare precedence:

```
aspect AandBandY {
  declare precedence: A, Y, B;
}
```

This composition code can be used with AandBandC shown above.

A problem with declare precedence is that AspectJ does not have a mechanism that overwrites existing declare precedence. Suppose that only when A, B, C, D, and X are used together, they must be executed in the order, A, B, C, D, and X. The following code declares the precedence order for that composition:

```
declare precedence: A, B, C, D, X;
```

However, the existing aspect, CandD cannot be compiled with this declare precedence and hence the advice of CandD is not reusable. This is because:

```
declare precedence: X, C, D;
```

from the aspect conflicts with the former declare precedence; one declares that C precedes X, but the other declares that X precedes C. Such a conflict causes a compile error in AspectJ, and to avoid the error, programmers must modify CandD.

István *et al.* proposed more expressive mechanisms for declaring precedence order [63]. They include two kinds of precedence constraints. $pre_{soft}(x, y)$ declares that aspect x precedes y, which is similar to declare precedence of AspectJ. On the other hand, $pre_{hard}(x, y)$ also gives higher precedence to x than y. Besides, y can be executed after x is executed at the same join point.

They also propose a mechanism that controls the execution of the aspects according to the result of one of the advices. $cond_{soft}(x, y)$ declares that aspect y can be executed only when aspect x returns true or x does not run. $cond_{hard}(x, y)$ allows y to be executed after x is executed and returns true.

Context-Aware Composition Rules [57, 58] extends AspectJ and introduced mechanisms to remove unnecessary advices or reorder conflicting advices at shared join point. The uniqueness of this mechanism is to specify rules for the precedence order among the advices for each join point. The rules are written within an advice-like construct, declare rule as shown in List. 2.19. It takes a pointcut for selecting join points when the rules inside are applied. The declare rule in the AspectComposition aspect has a advices() pointcut. It is a pointcut that captures join points when the advice given to

```
public aspect AspectComposition {
  declare rule AandB: advices(A.a, B.b) && if (...) {
    Prec(A.a, B.b);
  }
}

public aspect A {
  void around() a: ... {
    :
    proceed();
    :
  }
}

public aspect B {
  void around() b: ... {
    :
    proceed();
    :
  }
}
```

List. 2.19: Specifying precedence order in Context-Aware Composition Rules

its parameter is executed. To identify every advice, the language extension allows giving a unique name to every advice. Note that declare rule may have a if pointcut, programmers can change the rules depending on the dynamic context.

In the language extension, four kinds of rules are available. $Prec(x, y)$ declares advice x precedes y as declare precedence does. $First(x)/Last(x)$ executes x at the first/last, at their conflicting join point. $Ignore(x)$ prevents x from being executed. Programmers can specify combinations of those rules in the body of declare rule.

There are mechanisms for advice composition that are powerful but not enough composable. JAsCo [73] and POPART [25] provide meta-programming facilities, which allow writing meta-program executed at conflicting join points in order to control the execution order of the advices. List. 2.20 shows a typical meta-program available in those systems. This meta-code written as a subclass of ConflictManager gives precedence order between two logging aspects. The ConflictManager class is a part of the meta-APIs, and its onConflict() method is executed at the every join point when advices conflict. The method receives a list of meta-aspects (meta-objects representing aspects) corresponding to the conflicting advices. Programmers can specify the precedence order among the aspects by sorting that list. The onCon-

```
class Precedence1 extends ConflictManager {
  void onConflict(List<Aspect> aspects, JoinPoint jp) {
    if (containsItemWithName(aspects, "TraceLogging") &&
        containsItemWithName(aspects, "ArgumentLogging")) {
      Collections.sort(aspects, new Comparator<Aspect>() {
        int compare(Aspect a1, Aspect a2) {
          if (a1.getName().equals("TraceLogging") &&
              a2.getName().equals("ArgumentLogging")) {
            return HIGHER_PRECEDENCE;
          } else if (a1.getName().equals("ArgumentLogging") &&
                     a2.getName().equals("TraceLogging")) {
            return LOWER_PRECEDENCE;
          } else {
            // How can programmers implement here?
          }
        }
      });
}}}
```

List. 2.20: Giving precedence order by meta programming

flict() method sorts the list by using the sort() method of the standard Java API.

The comparator passed to the sort() method, however, must determine the total ordering of all the aspects conflicting at a join point. Unlike declare precedence, the comparator for a set of conflicting aspects cannot be composed from the comparator for its subsets. Thus, the composability of the comparators is low. Although it is possible to sort aspects multiple times by using different comparators and override the order sorted by the existing one, the composability is still, nevertheless, not sufficient.

Another mechanism that is powerful but not composable is to modify the behavior of existing aspects. Aspect-oriented languages such as Aspectual Mixin Layers [8, 9, 10] and JastAdd [29] support aspect refinements, which allow programmers to override existing advices from the outside of the aspects in the similar way to class refinements of AHEAD. They can avoid interference by modifying conflicting advices so that these advices work together.

OARTA [59], which is an extension to AspectJ, makes it possible for aspect to modify the pointcuts of the existing advices for the sake of advice composition. Programmers can prevent one of the conflicting aspect from being executed in a particular context by appending an if (...) pointcut to the advice. The following code in OARTA appends an if pointcut:

```
public aspect Composition {
  andpointcut: (* * Cache.*(*)): if (!Debug.isEnabled());
}
```

Here, andpointcut declaration takes a pattern selecting advices and a pointcut. It replaces the every pointcut of the selected advices with a conjunction of the original and the given pointcut. * * Cache.*(*) selects all advices of the Cache aspect. Assume that Cache has the following advice:

```
void around evalCache(): execution(* Expression+.eval(*)) {
  ...
}
```

The declaration replaces the pointcut of the evalCache() advice with:

```
execution(* Expression+.eval(*)) && if (!Debug.isEnabled())
```

In OARTA, orpointcut declarations are also available; it replaces the pointcut with a disjunction of the original and the given pointcut. Note that OARTA also extends the syntax of AspectJ to distinguish advices with their unique name.

However, since both the meta-programming and the aspect-modification approaches introduce new mechanisms for aspect composition, they cause another problem; those new mechanisms may also conflict with each other. For example, what happens if multiple conflict manager objects are available in shared join points? If two refinements extend the same advice, they may require manual composition. It is not clear how multiple andpointcut/orpointcut declarations modifying the same advice behave.

2.5.2.4 Advice-level Ordering

Although declare precedence in AspectJ gives precedence order among not advices but aspects, there are some combinations of advices that require *advice-level ordering* [55, 57, 58, 59]. List. 2.21 shows two aspects: ABasicFeature and ADependingFeature. Assume that programmers need to execute the first before advice of ABasicFeature due to dependency between the two aspects when the initializeApp() method is executed. The following code:

```
declare precedence: ABasicFeature, ADependingFeature;
```

allows executing the two before advices in such order. However, when finalizeApp() method is executed, they cannot execute the before advice of ADependingFeature first although the components should be finalized in the reverse order of the initialization. Note that if the two advices executed at the initializeApp() method can be implemented as after advices, the following declaration:

```
aspect ABasicFeature {
  before(): execution(void Application.initializeApp(void)) {
    ...
  }

  before(): execution(void Application.finalizeApp(void)) {
    ...
  }
}
.................................................................
aspect ADependingFeature {
  before(): execution(void Application.initializeApp(void)) {
    ...
  }

  before(): execution(void Application.finalizeApp(void)) {
    ...
  }
}
```

List. 2.21: Conflicting aspects requiring advice-level ordering

```
  declare precedence: ADependingFeature, ABasicFeature;
```

lets the advice of ADependingFeature run before the advice of ABasicFeature. The implementation of the base code might not allow to implement them as after advices.

Chapter 3

FeatureGluonJ

Implementing Feature Interactions with Generic Feature Modules
Fuminobu Takeyama and Shigeru Chiba.
In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Software Composition*, volume 8088 of *Lecture Notes in Computer Science*, pages 81–96, Springer Berlin Heidelberg, 2013

A challenge in FOP is the optional feature problem [48]. If multiple optional features interact with each other, any of feature modules for those features should not contain the code for the interaction since the code must be effective only when those interacting features are selected. Although a possible approach is separating such code into independent modules called *derivatives* [48, 54], the number of derivatives tends to be large as the numbers of features increases.

This chapter proposes a design principle to reduce the effort in implementing derivatives. A group of features indicating specialization of another feature is implemented by inheriting a common super feature module. The super feature module works as an interface among its sub-features and allows implementing a derivative in a reusable manner for every combination of sub-features. To demonstrate this principle, we developed a new FOP language, FeatureGluonJ. It provides a language construct called a generic feature module for the reusable implementations of derivatives as well as a feature-oriented module system supporting inheritance.

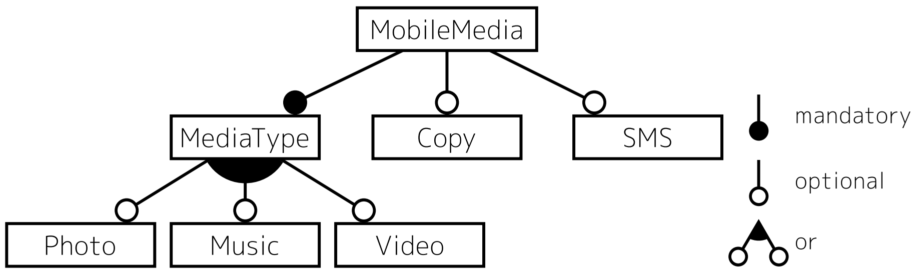

Fig. 3.1: The feature model of MobileMedia

3.1 The Optional Feature Problem in MobileMedia

This section explains the optional feature problem in a family of multimedia-management application for mobile devices, named the MobileMedia SPL. MobileMedia is widely used in the research community of SPLs. This chapter uses the six features from MobileMedia: MediaType, Photo, Music, Video, Copy, and SMS. Fig. 2.2 illustrates the feature model diagram consisting of features used in this example. Photo, Music, and Video are children of MediaType. The arc drawn among the children represents an *or*-relation and developers must select at least one feature from them if their parent is selected.

In FOP, a feature should be implemented as an independent feature module. If a feature is selected, the corresponding module is compiled together with other selected feature modules. In case of the original MobileMedia implemented in AspectJ if a feature is selected, aspects belonging to the feature are compiled and linked. If a feature is not selected for a product, its aspects do not affect the product.

However, a group of optional features may interact [20] with another group. For example, the Photo feature interacts with the Copy feature in MobileMedia. If both features are selected for a product, then the command for copying a photo is displayed in the pull-up menu of the screen, *i.e.,* window, showing that photo. This command should not be implemented in a feature module for Photo or Copy since the command is not activated unless Copy is selected. It should not be in a feature module for Copy since Copy may be selected without Photo. If so, the Copy feature module must not add the command to the menu. No matter where the command is implemented, Photo or Copy, the resulting code would cause undesirable dependence among optional features and lower the variability of a product line.

```
public aspect CopyAndPhoto {
  after(Image image) returning (PhotoViewScreen f):
      call(PhotoViewScreen.new(Image)) && args(image); {
    f.addCommand(new Command("Copy", Command.ITEM, 1));
}}
```

(a) CopyAndPhoto.aj

```
public aspect CopyAndMusic {
  pointcut initForm(PlayMediaScreen mediaScreen):
    execution(void PlayMediaScreen.initForm()) &&
    this(mediaScreen);

  before(PlayMediaScreen mediaScreen): initForm(mediaScreen) {
    mediaScreen.form.addCommand(
      new Command("Copy", Command.ITEM, 1));
}}
```

(b) CopyAndMusic.aj

List. 3.1: The derivatives for Copy and Photo/Music written in AspectJ

A more modular approach is to implement such interaction into an independent module called a *derivative*. A derivative is described as a normal feature module in AspectJ. We show a derivative for the combination of Photo and Copy in List. 3.1. This code is a part of the MobileMedia Lancaster[1], a MobileMedia implementation in AspectJ [33]. The CopyAndPhoto aspect implements the derivative. It has an advice executed after a constructor call for the PhotoViewScreen in order to add the command for copying a photo.

Note that feature interaction is often observed between feature groups. Suppose that two feature groups have n and m features. If a feature from one group interacts with one from the other, other pairs between the two groups will also interact with each other due to the similarity of features. Such interaction will require $n \times m$ derivatives in total. Furthermore, these derivatives will be similar to each other. They are redundant and should be merged into a single or only a few derivatives.

A group is often represented by a parent-children relation in a feature-model diagram. In Fig. 2.2, MobileMedia contains a group consisting of Photo, Music, and Video. We call this group the MediaType group named after the parent node. There is another group that the developers of the original MobileMedia did not recognize. It is a group consisting of Copy and SMS, which enable the users to send a photo shown on the screen by SMS. The two groups involve close interaction. Copy interacts with Music as well as Photo. The derivative for Copy and Music in List. 3.1 (b) is similar to

[1]We show simplified code for explanation. The original code is available from: `http://mobilemedia.sourceforge.net/`.

CopyPhoto in List. 3.1 (a). SMS also interacts with Photo, Music and Video[2]; if these features are selected, a command to send each medium must be added to the menu. Thus, MobileMedia requires 6 derivatives for the two groups.

The MediaType group is an extension point of MobileMedia. One of the goals of FOP is step-wise, *i.e.* incremental, development of large-scale software [18], and hence one of realistic development scenarios is adding a new media type as a new feature. Suppose that developers add plain-text documents as a new medium. Then they will have to implement derivatives for the combination of the plain-text feature and Copy and SMS.

3.2 Implementing Feature Interactions in FeatureGluonJ

Our feature-oriented programming language named *FeatureGluonJ* [3] provides language constructs to reduce redundancy of derivatives among feature groups. FeatureGluonJ is an extension of GluonJ, which is an aspect-oriented language based on Java. While GluonJ adds a new language construct called a *reviser* to Java, FeatureGluonJ also adds a generic feature module as well as a feature-oriented module system.

FeatureGluonJ provides an inheritance mechanism for feature modules. Features often make is-a relations [39, 42]. In MobileMedia, the Photo feature *is a* MediaType feature. Thus, in FeatureGluonJ, the Photo feature module, which is the implementation of Photo, is a sub-feature module of MediaType as shown in Fig. 2. It can not only add new classes but also redefine the classes contained in the MediaType feature module. The MediaType feature module works as a common interface to this feature group including Photo and Music. The interface represents which classes are commonly available in the feature group. The hierarchical implementations of features are not novel; they are provided by CaesarJ [12, 61, 62] and Object Teams [37, 39] as explained in Section 2.4.7. However, they are not studied in the context of modularity of feature modules [46, 72]; this paper focuses on how to use this inheritance mechanism to efficiently implement derivatives.

Another unique mechanism in FeatureGluonJ is a generic feature module. It is a feature module taking feature modules as parameters. Suppose that there are two feature modules. Then the derivatives for combinations

[2]The original MobileMedia does not support to send a music or a video by SMS. It is not clear that this limitation is caused by the optional feature problem.

[3]The FeatureGluonJ compiler is available from:
`http://www.csg.ci.i.u-tokyo.ac.jp/projects/fgj/`

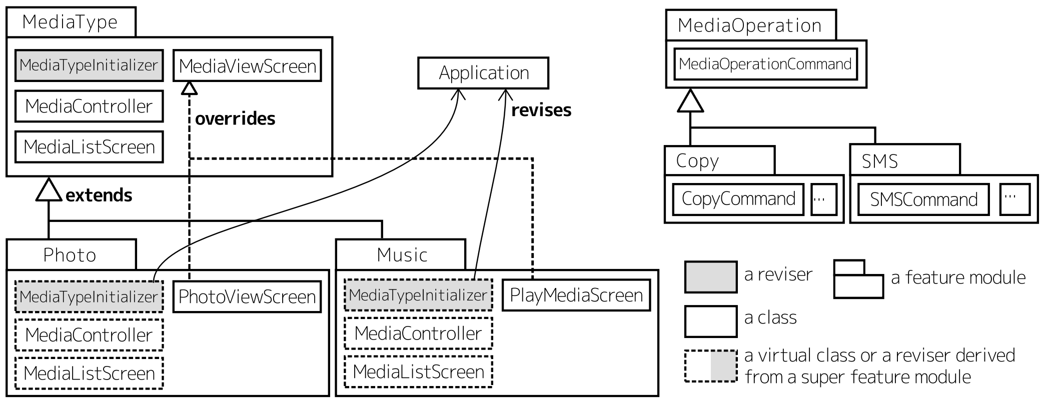

Fig. 3.2: An overview of the feature modules consisting MobileMedia in FeatureGluonJ

of their sub-feature modules are often almost identical. For example, in Fig. 2, the derivative for Photo and Copy is almost identical to the derivative for Music and SMS since they are for combinations between MediaType and MediaOperation. A generic feature module enables to describe such derivatives in a generic manner by using the interfaces specified by MediaType and MediaOperation. Note that the task of a typical derivative is to modify the classes in the feature modules that the derivative works for. These classes are often ones specified by the interfaces of the super feature modules such as MediaType and MediaOperation.

3.2.1 FeatureGluonJ

This section describes the overview of FeatureGluonJ to show how developers can implement an SPL. FeatureGluonJ provides a module system called feature modules. A feature module implements a feature and a derivative. It is represented by two constructs, a feature definition and a feature declaration. A feature definition is described in a separate file, and it defines a feature name and its relation to other features. List. 3.2 (a) defines the MediaType feature, which is an abstract feature for other features that are to support a media type. The body of this feature is empty in this example, but it may contain import feature declarations shown later.

A feature declaration is similar to a package declaration in Java. It is placed at the beginning of a source file and specifies that the classes and revisers in that source file belong to the feature modules. For example, the second lines of the List. 3.2 (b)–(e) are feature declarations. They declare that those three classes and a reviser belong to the MediaType feature. Note

```
abstract feature MediaType {
  // MediaType has an empty body.
}
```

List. 3.2: The definition of MediaType feature module

that each class declaration is separated into an independent file.

An abstract feature may represent a group made by *is-a* relationships; a sub-feature module of that abstract module defines a feature belonging to that group. Here, the Photo feature module is a sub-feature of MediaType, which is specified in the extends clause in List. 3.5 (a). Photo reuses the model-view-controller relation defined in MediaType.

After compilation of each feature, developers *select* feature modules needed for a product. Only the selected feature modules are linked together and included in the product. Which features are selected is given at link time. Note that they cannot select abstract features. If an abstract feature module like MediaType must be included in a product, the developers must select a sub-feature of that abstract feature.

To implement feature modules, FeatureGluonJ provides three kinds of class-extension mechanisms: subclasses, virtual classes, and revisers. The difference of those mechanisms is the range of effects. The first one is a normal subclass in Java and affects in the narrowest range. The extended behavior takes effect only when that subclass is explicitly instantiated.

The next class extension mechanism is virtual class overriding [31, 56]. Virtual classes enable to reuse a family of classes that refer to each other through their fields or new expressions. All classes in a feature module are virtualized in FeatureGluonJ; a reference to a virtual class is late-bound. A sub-feature module can implement a virtual class extending a virtual class in its super feature. It *overrides* the virtual class in the super feature with the new class, *i.e.*, class references to the overridden class is replaced with one to the new class. Virtual class overriding is effective only within the enclosing feature module, which includes its super feature module executed as a part of the sub-feature. It does not affect new expressions in the siblings of the sub-feature.

The syntax of virtual classes in FeatureGluonJ is different from other languages. To override a virtual class, developers must give a unique name to the new virtual class instead of the same name as the overridden class.[4] List. 3.5 (b) shows the PhotoViewScreen class that overrides MediaViewScreen of MediaType. An overridden class is specified by an overrides clause, placed

[4]Programmers can give the same name by implementing them in a different package.

```
package mobilemedia.controller;
feature MediaType;
import javax.microedition.lcdui.*;
import mobilemedia.ui.*;

public abstract class MediaController extends AbstractController {
  protected boolean handleCommand(Command command) {
    if (command == OPEN) {
      open(getSelected());
      return true;
    } else if (...) { ... }
  }

  protected void open(String s) {
    MediaListScreen scr = new MediaViewScreen(s);
    scr.setCommandListener(this);
    Display.setCurrent(scr);
}}
```

(b) MediaController.java

```
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public abstract class MediaViewScreen extends Canvas {
  protected void initScreen() {
    this.add(new Command("Close"));
}}
```

(c) MediaViewScreen.java

```
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public class MediaListScreen extends List {
  // forward command to controller if an item is selected
}
```

(d) MediaListScreen.java

```
package mobilemedia.main;
feature MediaType;
import mobilemedia.ui.MediaListScreen;
import mobilemedia.controller.MediaController;

class MediaTypeInitializer revises Application {
  private MediaListScreen screen;
  private MediaController controller;
  public void startApp() {
    controller = new MediaController();
    screen = new MediaListScreen(controller);
    super.startApp();
}}
```

(e) MediaTypeInitializer.java

List. 3.3: The classes of MediaType feature module

```
package mobilemedia.main;

public class Application {
  public static void main() {
    Application app = new Application();
    app.startApp();
  }

  public void startApp() {
    // initializing this MobileMedia application
  }
}
```

List. 3.4: The Application class, which has program entry point

```
feature Photo extends MediaType {}
```

(a) Photo.feature

```
feature Photo;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PhotoViewScreen overrides MediaViewScreen {
  public PhotoViewScreen(String s) {
    // : load selected image
  }
  protected void paint(Graphics g) {
    // : draw the selected photo on this screen.
}}
```

(b) PhotoViewScreen.java

List. 3.5: The Photo feature in FeatureGluonJ in the position of an extends clause. Another difference in syntax is that virtual classes cannot be syntactically nested, as separated into each class to a single file.

We adopt lightweight family polymorphism [68] to make the semantics and the type system simple by avoiding dependent types. A feature module cannot be instantiated dynamically. It can be regarded as a singleton object instantiated when it is selected at link time.

The third mechanism is a reviser. A reviser can extend any class in a product; the extended behavior affects globally.[5] A reviser plays a similar role to the one of aspect in AspectJ; its code overrides classes appearing in any other feature module. The class-like mechanism with a keyword revises in List. 3.2 (e) is a reviser. The reviser has the startApp() method, which replaces the startApp() method in the class specified in its revises clause, *i.e.*, the Application class in List. 3.2 (f). Whenever the startApp() method is called on an Application object, the reviser's startApp() method is first executed. By calling super.startApp(), the replaced method is executed. A reviser can also add new fields to an existing class. The reviser in List. 3.2 (e) adds the two fields, screen and controller, to the Application class.

Revisers in a feature module are also virtualized. A feature module derives revisers as well as virtual classes from its super feature to reuse structure made by revisers and classes defined there. The Photo feature module in List. 3.5 does not contain any classes and revisers except the PhotoViewScreen class, but it also derives virtual classes such as MediaController and the MediaTypeInitializer reviser from MediaType (Fig. 3.2). A reviser will be executed only if a feature enclosing or deriving that reviser is selected. If Photo is selected, MediaTypeInitializer derived by the Photo is executed in the context of Photo. Within this MediaTypeInitializer, new MediaListScreen(...) will create an object of the class derived by Photo. The expression new MediaViewScreen() in List. 3.2 (b) on that object will instantiate PhotoViewScreen. The MediaTypeInitializer reviser might be derived by siblings of Photo. Suppose the Music feature module in List. 3.6 is implemented in the same way as the Photo. If both Photo and Music are selected, two *copies* of MediaTypeInitializer will be executed in the startApp() method but in different contexts.

These class extension mechanisms provided by FeatureGluonJ are an abstraction of the factory method pattern. For virtual-class overriding, each selected feature has its own factory. It receives a name of virtual class and returns an object of the class overriding the given class. Every new expression can be considered as a factory method call. Each virtual class has a reference to such factories. When a factory creates an object, it assigns itself

[5]GluonJ does not support global modifications [11].

```
feature Music extends MediaType {}
```

(a) Music.feature

```
feature Music;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PlayMediaScreen overrides MediaViewScreen {
  public PlayMediaScreen(String s) {
    // : load selected image
  }
  protected void paint(Graphics g) {
    // : draw music player
}}
```

(b) PlayMediaScreen.java

List. 3.6: The Music feature module in FeatureGluonJ

to the object. A factory used in a reviser is given by the linker when its feature module is selected. This factory is one used for virtual classes in the feature module containing or deriving that reviser.

A reviser, on the other hand, can be emulated by a factory shared among all the classes in a product. If a class given to a new expression is not a virtual class, the global factory will create an object. This global factory is also used inside of a factory for each feature. Note that it is unrealistic to manually implement factory methods for every class. Moreover, a factory method pattern degrades type safety.

3.2.2 Derivatives in FeatureGluonJ

FeatureGluonJ provides two other constructs for referring to virtual classes in other modules. One is import feature declarations. To make coupling of other features explicit, developers have to declare the features required in a derivative by import feature declarations. These declarations just open the visibility scope to virtual classes of imported features. Note that when a feature module with import feature is selected, the imported features are also selected. Since an abstract feature module is never selected, it cannot be imported.

An import feature declaration is described in the body of a feature declaration. List. 3.7 (a) contains two import feature declarations. An identifier after a colon indicates a feature module used in this module. The left one before the colon is an alias to the imported feature module. Then *feature-qualified access* is available as a reference to a virtual class of the imported feature

```
feature CopyPhoto {
  import feature c: Copy;
  import feature f: Photo;
}
```

(a) CopyPhoto.feature

```
package mobilemedia.copy;
feature CopyPhoto;
import mobilemedia.ui.*;

class AddCopyToPhoto revises f::PhotoViewScreen {
  protected void initScreen() {
    super.initScreen();
    this.addCommand(new c::CopyCommand());
}}
```

(b) AddCopyToPhoto.java

List. 3.7: The derivative between Copy and Photo rewritten from List. 3.1

module. The access is represented by a :: operator. The left of :: must be an alias declared in the feature module and the right of :: is the name of a virtual class in the feature module expressed by the alias. For example, List. 3.7 implements a derivative straightforwardly rewritten from List. 3.1. In the AddCopyToPhoto, p::PhotoViewScreen refers to the PhotoViewScreen class in the Photo feature since p is an alias of Photo. The reviser extends PhotoViewScreen and adds a command for copying a medium, which is now represented by the CopyCommand class in the Copy feature module shown in List. 3.8 (c) and (d).

The reason FeatureGluonJ enforces programmers to use feature-qualified access is that multiple feature modules may contain virtual classes with the same name if they extend the same module. For example, both of Photo and Music contains the MediaController class derived from MediaType, which are distinguished by aliases.

3.2.3 Generic Feature Modules

We found that if features are implemented by a feature module with an appropriate interface, most derivatives can be implemented by a special feature module that takes the name of required sub-features as parameters. FeatureGluonJ provides a *generic feature module*, which is a reusable feature module to implement derivatives among features extending common feature modules. The Copy feature and the SMS feature, which is not shown but implemented in the same way in List. 3.8, are sub-feature modules of MediaOperation in List. 3.8 (a) and (b). Now the generic derivative among

```
abstract feature MediaOperation {}
```

(a) MediaOperation.java

```
package mobilemedia.ui;
feature MediaOperation;
import javax.microedition.lcdui.Command;

public abstract class MediaOperationCommand extends Command {
  public MediaOperationCommand(String labelText) {
    super(labelText, ...);
}}
```

(b) MediaOperationCommand.java

```
feature Copy extends MediaOperation {}
```

(c) Copy.feature

```
package mobilemedia.ui;
feature Copy;

public class CopyCommand overrides MediaOperationCommand {
  public CopyCommand(String labelText) {
    super(labelText);
}}
```

(d) CopyCommand.java

```
package mobilemedia.controller;
feature Copy;
import mobilemedia.ui.*;

public class CopyController revises MediaController {
  protected boolean handleCommand(Command command) {
    if (command instanceof CopyCommand) {
      :
    } else { return super.handleCommand(); }
}}
```

(e) CopyController.java

List. 3.8: The Copy feature module implemented by extending the MediaOperation

```
abstract feature MediaOperationMediaType {
  abstract import feature o: MediaOperation;
  abstract import feature t: MediaType;
}
```

(a) MediaOperationMediaType.java

```
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

class AddCommandToMediaType revises t::MediaViewScreen {
  protected void initForm() {
    super.initForm();
    this.addCommand(new o::MediaOperationCommand());
}}
```

(b) AddCommandToMediaType.java

List. 3.9: A generic derivative implementing common part of derivatives among MediaOperation and MediaType

sub-features of MediaOperation and MediaType takes sub-features of those modules as parameters and behaves for a derivative among the given features.

A generic feature module is represented by an abstract feature module. It may contains import feature declarations with an abstract keyword. An alias defined by this abstract import feature works as a parameter; the alias is late-bound to a concrete module, which must be a sub-feature of one apparently assigned to the alias. An abstract import feature may import an abstract feature module. List. 3.9 shows a generic feature modules for derivatives between sub-features of MediaType and MediaOperation. The generic feature modules contains two abstract import declarations that import FileOperation and MediaType with the aliases, t and o, respectively.

The AddCommandToMediaType reviser in List. 3.9 (b) is almost the same to AddCopyToPhoto expecting for the specific parts to Copy and Photo. The reviser extends a class indicated by t::MediaViewScreen and adds command indicated by o::MediaOperationCommand. Since t and o must be bound to sub-features of the imported features, it is ensured that they provide virtual classes overriding MediaOperationCommand and MediaViewScreen. If those aliases are bound to Copy and Photo, this reviser is semantically the same as AddCopyToPhoto.

A feature module can extend a generic feature module and bound aliases declared in its super feature to concrete feature modules. If a feature module imports another feature module with the same alias as one used in its super feature, the alias is bound to that feature module also in the super feature.

```
feature CopyPhoto extends MediaOperationMediaType {
  import feature o: Copy;
  import feature t: Photo;
}
```

List. 3.10: Another derivative for Copy and Photo extending the generic derivative

Suppose another implementation of the CopyPhoto feature module, which is implemented by in List. 3.10 by extending the generic feature module in List. 3.9. It assigns o and t to Copy and Photo respectively.

3.2.4 A Composition Language for Trivial Feature Modules

If each interacting feature is properly implemented, a derivative may not contain its own reviser nor class; we call such derivative is trivial. The derivatives among MediaType and MediaOperation including CopyPhoto in List. 3.10 are trivial. This is because operations such as copying a medium or sending it by SMS are reduced to operations against streams of bytes.

FeatureGluonJ also provides a composition language to define such trivial derivatives implicitly. It includes a construct defines forevery. If our linker interprets a feature module with defines forevery, it defines sub-features of this derivative automatically at linking time. defines forevery receives one or more aliases to feature modules. If the given alias is declared in an abstract import feature, it represents a set of its sub-features that are selected for the linker. The linker will define and select sub-feature modules for every combination from each given set. Let $a_1, a_2, .., a_n$ be aliases given to the defines forevery and

$$S_i = \{f | f \in Sub(a_i) \cap f \ is \ selected\}$$

where $Sub(a)$ is a function returning the set of the sub-features that might be bound to a. A sub-derivative is created for each element of $S_1 \times S_2 \times \ldots \times S_n$. If a is an alias of a concrete feature, $Sub(a)$ returns the set containing the concrete feature only.

List. 3.11 shows derivatives for sub-features of MediaOperation and MediaType including derivative between Copy and Photo. The defines forevery clause allows programmers to omit concrete feature modules such as one in List. 3.10. Even when developers add a new feature for a new media type, they would not implement new derivatives if this generic derivative is applicable for the new feature. Otherwise, programmers would implement extra

```
feature MediaOperationMediaType defines forevery(o, t) {
  abstract import feature o: MediaOperation;
  abstract import feature t: MediaType;
}
```

(a) MediaTypeFileOp.java

```
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

class AddCommandToMediaType revises t::MediaViewScreen {
  protected void initForm() {
    super.initForm();
    this.addCommand(new o::MediaOperationCommand());
}}
```

(b) AddCommandToMediaType.java

List. 3.11: Our final version of derivatives among MediaOperation and MediaType by defines forevery

behavior for the specific combinations of feature modules as a new derivative.

3.2.5 Discussion

We discuss on the limitations of our language. Unfortunately, all derivatives do not become trivial after refactoring. Some derivatives are essential, which must be implemented manually. We can find essential derivatives in the expression product line [46]. Derivatives among a feature for an operator and feature for evaluating expressions is unique to each combination of features. If the feature has redundant parts, FeatureGluonJ allows to reuse it with a generic-feature module.

Although inheritance allow us to implement generic derivatives, it may cause extra effort to implement an SPL. We introduced the common super class between PhotoViewScreen and PlayMediaScreen. As shown in List. 3.1 (a) and (b), the original derivative uses different methods to add their commands to the menus; in Photo, it is the constructor of PhotoViewScreen, but in Music, it is the initForm() method. We add the common super class MediaViewScreen and its initScreen() method in List. 3.2 (c) to unify those methods among both features. We also defines Copy and SMS by extending MediaOperationCommand to make the derivatives trivial. The implementations of these feature modules are in a sense composition aware.

Our observation is that whether or not we should implement each feature considering composition is a design decision. The MediaType group and the MediaOperation group are extension points; in other words, a new feature will be added to these groups in the future. The cost of making these features composable is much lower than a large number of derivatives.

3.3 A Case Study

While investigating the original MobileMedia, we also found three other implementation problems to make classes and aspects reusable within the confines of the existing FOP approach, namely AspectJ. This section shows how we can address these problems in FeatureGluonJ.

Refactoring aspects into virtual classes

In the original implementation of MobileMedia, the specialized features contain aspects destructively extending the class commonly used among the features instead of making its subclass for each of those features. These conflicting aspects cause a maintenance problem. The controllers for Photo and Music features were implemented by aspects that destructively extend the common class, MediaController as shown in List. 3.12. Suppose that we have two instances of MediaController for Photo and Music. Both of those aspects are always executed regardless of the actual feature of the instance of MediaController. Thereby, the aspects distinguish the current feature by the label of a pushed button. The aspect for Photo opens its photo viewer screen only when the label is "View". The label must be unique among the features providing types of medium. If the Video feature used "Play" for its label, the Music and Video features would conflict when they are selected together in a product.

FeatureGluonJ allows implementing those controllers in virtual classes without redundancy. The redundant aspects above are implemented by the MediaListController in List. 3.3. The code snippets used only for Photo are implemented in PhotoController in List. 3.5 overriding MediaListController. This overriding does not affect Music.

Rewriting || pointcut operator to a derivative

A || pointcut is useful to extend different methods by a single advice in AspectJ, but using || pointcut might make aspects fragile and break grouping of related code. Here, we introduce the Sorting feature, which enables to sort a list of media according to how many times they were shown or played. It is implemented as the Sorting aspect in List. 3.13. It contains an after advice

```
class MediaController {
  void handleCommand(Command command) {
    // invoked when buttons on screens are pushed.
  }
}
...............................................................................
aspect PhotoAspect {
  boolean around(MediaController cont, Command command):
    execution(boolean MediaController.handleCommand(Command)) &&
    this(cont) && args(command) {
  if (proceed()) {
    return true;
  } else if (command.getLabel().equals("View")) {
    showImage(getSelected());
    return true;
  } else {
    return false;
  }
}
...............................................................................
aspect MusicAspect {
  boolean around(MediaController cont, Command command):
    execution(boolean MediaController.handleCommand(Command)) &&
    this(cont) && args(command) {
  if (proceed()) {
    return true;
  } else if if (command.getLabel().equals("Play")) {
    playMusic(getSelected());
    return true;
  } else {
    return false;
  }
}
```

List. 3.12: Dynamic dispatch with values

```
aspect Sorting {
  after(String selected):
    (execution(void MediaController.show(String) ||
     execution(boolean MediaController.play(String)))) &&
    args(selected) {
    // : increment a counter of a selected item
  }
}
```

List. 3.13: Reusing the Sorting aspect with a || pointcut

```
feature SortingMediaType defines forevery(m, s) {
  abstract import feature m: MediaType;
  import feature s: Sorting;
}
.............................................................................
feature SortingMediaType;

reviser IncrementsCounter extends s::MediaController {
  void open(String selected) {
    super.open();
    // : increment a counter of selected item
  }
}
```

List. 3.14: The Sorting feature in FeatureGluonJ

that extends show() method and the play() method so that it can counts the number of times of opening media. The || pointcuts allows extending these two methods by the single advice. Even though the advice depends on two features, show() in Photo and play() in Music, it still works when only either of those features is selected. This is because AspectJ permits to specify undefined methods to execution pointcuts; such pointcuts does not much anywhere. These semantics of AspectJ is double-edged; even if programmers forget to modify the pointcut when the play() method is renamed, the aspect is still valid. Furthermore, programmers must append a new || pointcut operator to the aspect when they implement a new feature like Video.

The advice should be separated into a derivative between Sorting and Photo and one between Sorting and Music. In FeatureGluonJ, the advice can be separated into a derivative, SortingMediaType as shown in List. 3.14. The implementation in FeatureGluonJ unifies the show() and the play() methods into open(), and subclasses of MediaController overrides that method for Photo and Music. The defines clause of SortingMediaType applies the IncrementCounter reviser to classes overriding MediaController according to

```
feature Video extends Music {}
..............................................................................
feature Video;

class VideoListScreen overrides MusicListScreen {
  : //music-specific codes
  void paintItem(Graphics g, MediaData m) {
    // draw a thumbnail of a video
  }
}

class VideoController overrides MusicController {...}
class VideoPlayerScreen overrides MusicPlayerScreen {...}
```

List. 3.15: The Video feature implemented by extending Music

selected features.[6] Programmers would not need to modify the derivative to add a new feature providing a type of medium.

Resolving redundancy by inheriting a non-abstract feature

The implementation of the Video feature is also redundant since it equips similar interfaces to the Music feature. In FeatureGluonJ, programmers can implement the Video feature intuitively in FeatureGluonJ by extending the Music feature as shown in List. 3.15. The MediaTypeInitializer reviser in the MediaType feature is also executed in the context of the Video feature. The controller for Video can be implemented by overriding and reusing the MusicController class.

3.4 Comparison to the Existing Approaches

Feature-oriented programming

Most of feature-oriented approaches such as AHEAD Tool Suite [18], FeatureHouse [7], and Delta-oriented programming [70] are based on the idea that a feature is represented as a layer comprising of classes and refinements. A product is simply represented as a layer stacked linearly [17]. Those approaches do not provide mechanisms for implementing features including derivatives by reusing other modules.

[6]A reviser in a feature module with a defines clause can be considered as a global modification [11].

Although FeatureGluonJ does not provide tools for selecting features or algebra models, the existing approaches can be used for feature modules implemented in FeatureGluonJ. This is because a concrete feature module defined by extending another feature module still a set of classes and refinements.

Virtual classes

As explained in Section 2.4.6, virtual classes are capable of implementing feature-oriented product lines without using virtual revisers. However, the implementations involve code redundancy. Suppose that we implement the MobileMedia in some language with multilevel-nested virtual classes, variant path types [40], and CaesarJ-like multiple inheritance. See the PhotoFeature class in List. 3.16. It contains two first-level virtual classes, Application and Photo. Application is used like a reviser; it overrides another Application class in the Base class, which is the super feature of the PhotoFeature. ^This is a construct for variant path types. It expresses the outer class of an actual class. ^This.Photo.Controller is a virtual class reference to the PhotoFeature.Photo.Controller class. Since this pseudo language is a lightweight approach, it is not needed to instantiate an outer class before using its nested classes. The Photo class contains virtual classes used for this feature. These virtual classes are directly corresponds to the virtual classes in List. 3.5. The common code snippets between Photo and Music are extracted into the MediaType class of the Base class.

The two Application classes in PhotoFeature and MusicFeature are redundant although this problem can be resolved by a virtual reviser in FeatureGluonJ. We could not resolve this redundancy even if we wrote:

```
new ^This.MediaType.Controller();
```

at the place (A) in List. 3.16 and removed the two startApp() methods in PhotoFeature and MusicFeature. Since no class overrides MediaType (it is only extended), the virtual reference ^This.MediaType.Controller does not refer to Photo.Controller or Music.Controller. If the Application class is moved to the place (B) within the body of MediaType, then that virtual reference will work. However, this produces two distinct versions of the Application class, one for Photo and the other for Music. The startApp() method does not create both Photo's and Music's controllers. This behavior is different from our original intention. To fix this, we need a mechanism to tell the two versions of Application are merged.

Package templates [14, 52] allows merging the two versions of Application into one Application class while a package instantiates templates implement-

```
class Base {
  class Application {
    void startApp() {
      // (A)
    }
  }

  abstract class MediaType {
    class Controller { ... }
    // (B)
  }
}
..............................................................................
class PhotoFeature extends Base {
  class Application {
    private ^This.Controller cont;
    void startApp() {
      new ^This.Photo.Controller();
      super.startApp();
    }
  }

  class Photo extends MediaType {
    class Controller { ... }
    // :
  }
}
..............................................................................
class MusicFeature extends Base {
  class Application {
    private ^This.Controller cont;
    void startApp() {
      new ^This.Music.Controller();
      super.startApp();
    }
  }

  class Music extends MediaType {
    class Controller { ... }
    // :
  }
}
..............................................................................
class PhotoMusic extends PhotoFeature, MusicFeature {
  public static void main(String[] args) {
    (new Application()).startApp();
  }
}
```

List. 3.16: A MobileMedia implementation with the second-level nested classes ing Photo and Music. However, as shown in Section 2.4.9, programmers must implement a package for each product. Furthermore, they need to specify how the Application classes and their methods are merged in the package.

Dependent classes

Our reusable derivatives implemented by abstract import feature are similar to dependent classes [36] and their lightweight variation [41]. A dependent class is a parameterized class that behaves polymorphically depending on its parameters. On the other hand, a virtual class can be regarded as a class that behaves polymorphically depending on a single implicit parameter, which is the actual class of its outer class. In FeatureGluonJ, behavior of virtual classes depends on imported features as well as their actual feature modules.

Dependent classes are not suitable for implementing features because they do not straightforwardly fit to feature modules. Moreover, dependent classes do not resolve the redundancy problem common to other virtual-class approaches. Dependent revisers could resolve that problem, but they are virtual revisers in FeatureGluonJ.

Hierarchical implementations of feature modules

FeatureGluonJ offers mechanisms for hierarchical implementations of feature modules as CaesarJ [12] and Object Teams [37] do. The difference from those languages is language support for generic derivatives. Object Teams provides a dependent team, which behaves polymorphically depending on a given instance of a team. The origin of the dependent team is a dependent class [36]. However current specification of Object Teams [38] does not allow teams dependent to multiple teams. Dependent teams hence cannot be used for derivatives among groups. Those languages may be able to demonstrate our design principle by their first-class features although they require boilerplate code for instantiating such features for each product.

Destructive extensions with scoping

Virtual class overriding can be regarded as destructive extension that is effective only within limited scope. There is literature on scope mechanisms for destructive extensions. ClassBox/J [19] extends Java's package mechanism and provides a destructive extension construct like a reviser. A package can import classes from another package and destructively extends them. The effect of the extension is limited within that package. The semantics of ClassBox/J is much similar to lightweight virtual classes; packages corresponds classes enclosing virtual classes, and importing classes from another package can be considered as inheriting them from another class. However, ClassBox/J is not designed so that different extensions to the same class can be used together in a package. Moreover, since it does not provide destructive extensions to classes of other packages, it also involves the redundancy problem common to virtual classes.

Open modules [4, 64] are mechanisms for preventing a specific aspect from being executed on a class. Since open modules uses AspectJ's pointcut language to specify conditions for executing an aspect, it controls the execution of aspects according to the dynamic context obtained by cflow or if pointcuts. In order to emulate family polymorphism, programmers must manually embed a value for distinguishing current feature to each object.

Other approaches

In annotation-based approaches for SPLs, code regions implementing a feature is annotated with syntactical blocks, #ifdef and #endif. CIDE [45] represents such annotations as background colors of code snippets on text editors of integrated development environment. Code snippets with the same color means they belong to the same feature. It also enables virtual separation of features by hiding unnecessary features from the editors. Since feature interaction is indicated by intersections of the annotated regions [16], programmers can understand interactions more intuitively than derivative modules. However, the reusability of feature modules and derivatives is not clear in annotation based approach. Our language-based approach enables to reuse code snippets multiple times for different features in a single product.

3.5 Implementation

We have implemented a FeatureGluonJ compiler built on the existing GluonJ compiler and weaver[7]. Our compiler consists of three tools as illustrated in Figure 3.3. We have modified the GluonJ compiler so that it can deal with our extended syntax and semantics, and append meta-information necessary for the subsequent process. We also introduced a new tool, the *Feature Configurator*, which generates classes and revisers depending on selected features. The FeatureGluonJ compiler uses GluonJ's weaver as it is. We have implemented the compiler by writing about 2500 lines of code as shown in

[7]http://www.csg.ci.i.u-tokyo.ac.jp/projects/gluonj/

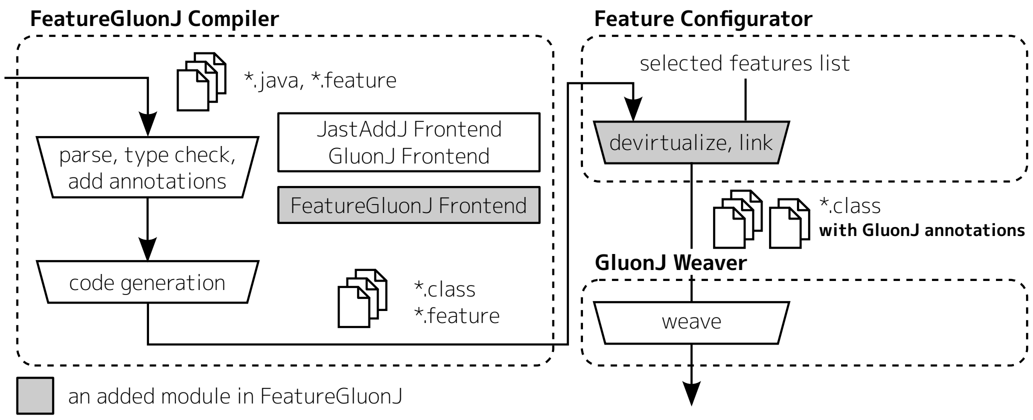

Fig. 3.3: An overview of FeatureGluonJ implementation

	Physical lines of code
JastAdd Aspects added to GluonJ	722
AST definitions and parsing rules	103
Configurator	650
The feature definition parser, etc.	992

Table 3.1: The code size of components of FeatureGluonJ

Table 3.1, due to JastAdd and the Javassist bytecode manipulation library [21] used for Feature Configurator.

The compilers of both FeatureGluonJ and GluonJ have been developed by using the JastAdd compiler-compiler framework [29], which supports extensible implementation of compilers. It provides an aspect-oriented system to implement manipulation of abstract syntax. A Java compiler implemented by JastAdd, JastAddJ, consists of syntax rules, AST classes and aspects that execute name analysis, type checker, and code generation. Not only to group related code, introducing a new aspect allows to customize an existing compilation process without modifying the original code. Since GluonJ is an extension to Java, its compiler is implemented by combining syntax rules and an aspect unique to it with ones of JastAddJ. The compiler translates revisers into classes with the @Reviser annotations used for annotation-based representation of GluonJ.

To implement our compiler, we added several aspects to the GluonJ compiler. The aspects implement name analysis and type checking of virtual classes. If they do not detect a type error, they translate construct of FeatureGluonJ into normal Java classes with annotations similarly to GluonJ.

The compiler translates a feature declaration of each file into the @Feature annotations and appends them to all the classes in the file. It also converts a virtual class with overrides into a normal subclass of its overridden class and appends the @Overrides annotation to the subclass. Since Java 7 does not support appending annotations to a type access, Java annotations cannot express the feature-qualified accesses. Thereby, we adopt an approach that mangles (*i.e.*, encodes) the class names referred by using feature-qualified accesses.

The FeatureGluonJ Configurator is a tool that generates classes and revisers of the resulting product according to the selected features for that product; it gets rid of virtual classes of unused features. It also automatically implements sub-features of feature modules with abstract import feature, applying the selected feature modules to the aliases, for every combination of the selected features. The output of this tool is class files with GluonJ annotations, which are passed to the weaver and revisers are to be woven into classes.

We employed an approach that devirtualizes virtual classes and virtual revisers into non-virtual ones instead of emulating them by using the factory method pattern. This is because all the virtual references can be resolved statically. The procedure of the devirtualization is as the following:

1. The tool duplicates virtual classes for each actual feature. The input of the tool is FeatureGluonJ classes represented by the annotations as shown in List. 3.17 (a). Suppose that we have selected the Photo and Music features for the MobileMedia. The tool makes copies of all the virtual classes and revisers defined in their super feature module, MediaType, for them. For example, we will have two copies of the MediaViewScreen class of MediaType; one of them is for Photo and it is represented as Photo$$MediaViewScreen in List. 3.17 (b); the other is for Music. Copies of the MediaController class and the MediaTypeInitializer reviser will be also generated for the selected features.

2. Next, the configurator processes virtual classes in the sub-features. A virtual class overriding another class is translated into a subclass of the copy of the overridden class generated for the sub-feature. In MobileMedia, PhotoViewer is a virtual class of the Photo feature and overrides MediaViewScreen. The configurator changes PhotoViewScreen to a subclass of Photo's copy of MediaViewScreen; in List. 3.17 (b), the copy is Photo$$MediaViewScreen.

3. Then, all the virtual class references including ones in new expressions are resolved by the configurator. References to overridden classes in the

```
@Feature("MediaType")
class MediaViewScreen extends Canvas { ... }

@Feature("MediaType")
class MediaController extends AbstractController {
  ... new MediaViewScreen(...);
}

@Feature("MediaType") @Reviser
class MediaTypeInitializer extends Application {
  ... new MediaController();
}

@Feature("Photo") @Overrides
class PhotoViewScreen extends MediaViewScreen { ... }
```

(a) The classes of the MobileMedia given to the configuration tool

```
class Photo$$MediaViewScreen extends Canvas { ... }

class Photo$$MediaController extends AbstractController {
  ... new MediaViewScreen(...);
}

@Reviser
class Photo$$MediaTypeInitializer extends Application {
  ... new MediaController();
}

@Feature("Photo") @Overrides
class PhotoViewScreen extends Photo$$MediaViewScreen { ... }
```

(b) The classes after step 1 and step 2

```
class Photo$$MediaViewScreen extends Canvas { ... }

class Photo$$MediaController extends AbstractController {
  ... new PhotoViewScreen(...);
}

@Reviser
class Photo$$MediaTypeInitializer extends Application {
  ... new Photo$$MediaController();
}

class PhotoViewScreen extends Photo$$MediaViewScreen { ... }
```

(c) The classes after step 3

List. 3.17: The class transformation by the Feature Configuration Tool copies of virtual classes and revisers for a sub-feature are replaced with references to the overriding class defined in that sub-feature. In the Photo's copy of MediaController (Photo$$MediaController in List. 3.17 (c)), it replaces all the references to MediaViewScreen with ones to PhotoViewScreen. In the Music's copy, all the references to that class are replaced with ones to PlayMediaScreen. All the virtual class references in the copied reviser are also replaced.

A limitation of this approach is the size of binary code. Numbers of copies of virtual classes are included in the resulting products. Using the factory method pattern for instantiating virtual classes allows avoiding duplicating virtual classes.

3.6 Summary

In this chapter, we have shown how derivatives among feature groups are implemented efficiently in FeatureGluonJ. First, designing feature modules hierarchically makes features modular, and it is important for implementing derivatives. FeatureGluonJ facilitates to implement generic derivatives among feature groups represented by the inheritance. Such derivatives are written by using super features as interfaces.

Chapter 4

Airia

An Advice for Advice Composition in AspectJ
Fuminobu Takeyama and Shigeru Chiba.
In Benoît Baudry, and Eric Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 122–137, Springer Berlin Heidelberg, 2010

In aspect-oriented programming, an application is composed of classes and aspects. The aspects often conflict with each other, *i.e.* they are woven into the same join point. Since some aspects implement concerns interacting with others, they are often woven into the same join point. These conflicting advices sometimes interfere with each other and hence cause unexpected behavior. This unexpected behavior is called *advice interference.* This has been a well-known problem in the research community of aspect-oriented programming.

A naive and troublesome solution for avoiding interference is implementing advices to cooperatively work with other conflicting advices. Such aspects will be linearized by declare precedence and proceed() calls so that they will be executed one by one and show consistent composed behavior. However, this is not a desirable solution since programmers have to be aware of the composition with other advices when they are implementing an individual aspect. The code for consistent composition crosscuts over the bodies of conflicting advices. Although meta-programming allows programmers to describe composition code separately from conflicting advices, the existing meta-programming approaches do not provide sufficient functionality for composition.

This chapter takes a new approach to solve this aspect-interference problem. This approach enforces programmers to explicitly describe the composed behavior of conflicting advices. A compiler statically check this. The code implementing that behavior is instead executed at the join points when the advices conflict with each other. To make this approach feasible, this chapter also proposes a new language construct named *resolver* and new language *Airia*, which is an extension of AspectJ [50] but provides resolvers. A resolver allows programmers to flexibly combine existing advices to easily implement the composed behavior. For example, declare precedence of AspectJ is a simple mechanism for advice composition. It statically linearizes advices in specified order. On the other hand, a resolver can dynamically give appropriate precedence to conflicting advices and invoke them. It can also remove some of them and introduce new advices at the join points as helpers. Our language Airia thereby gives better composability to advices. In Airia, advices are not only simply combined but also overridden if they are not necessary to implement the composed behavior of them.

Furthermore, since a resolver is a special kind of advice, it is also the first-class entity that is composable by using the same language construct — a resolver. If a resolver implementing the composed behavior of conflicting advices also conflicts with other advices, then programmers can describe another resolver that implements the composed behavior of the former resolver and the other advices. Thus, the resolver hierarchically implements the composed behavior of all the conflicting advices. It is also possible to describe a resolver that implements the composed behavior of multiple other resolvers. Combining resolvers may cause inconsistent or ambiguous specifications of precedence order. Our compiler provides compile time check of conflict resolution. If the resolvers do not define consistent composed behavior, it reports a compile error.

In summary, the contributions of this chapter are the followings:

- We propose a new advice composition mechanism for resolving advice interference. It gives flexible composability to advices.
- Since our new advice composition mechanism is a special kind of advice, it is also composable by using the same mechanism. This allows hierarchical composition.
- Our compiler statically checks the composition is consistent.

```
public class Expression extends ASTNode {...}

public class Plus extends Expression {
  private Expression left;
  private Expression right;

  public Plus(Expression left, Expression right) {...}

  public Expression getLeft() {...}
  public Expression getRight() {...}
  // :
}

public class Constant extends Expression {
  Object value;
  public Constant(Object value) {
    this.value = value;
}}

aspect Evaluation {
  public Object Expression.eval() {
    return null;
}}
```

List. 4.1: Classes representing the AST

4.1 An Example of Aspect Interference

We first show an example of aspect interference that is unavoidable with existing composition mechanism.

A Simple Interpreter

We present a simple interpreter with a binary operator +, which is written in AspectJ. List. 4.1 shows classes representing AST (Abstract Syntax Tree) nodes. The Plus class expresses a binary operator +. It has two fields, left and right, representing its operands and it extends the Expression class. We declare a method for evaluating an AST in the EvaluationAspect aspect. Since our interpreter currently does not support any data types, this aspect appends an empty eval() method to the Expression class by an inter-type declaration.

Now a programmer extends the interpreter to support integer values. She does not have to modify the existing classes. She has only to write a new aspect shown in List. 4.2. The around advice in the IntegerAspect aspect is invoked when the Plus.eval() method is executed; it returns a summation of the two operands. The following code makes an AST representing 1 + 2.

```
aspect IntegerAspect {
  Object around(Plus t):
      target(t) && execution(Object Plus.eval()) {
    Object lvalue = t.getLeft().eval();
    Object rvalue = t.getRight().eval();
    if (lvalue instanceof Integer && rvalue instanceof Integer) {
      // not for composition
      return (Integer)lvalue + (Integer)rvalue;
    } else {
      return proceed(t);
}}}
```

List. 4.2: An aspect for integer values

```
aspect IntegerAspect {
  Object around(Plus t):
      target(t) && execution(Object Plus.eval()) {
    return (Integer)t.getLeft() + (Integer)t.getRight;
}}
```

List. 4.3: An aspect that does not consider other data types

When e.eval() is executed on this tree, it returns 3:

```
  Expression e = new Plus(new Constant(1), new Constant(2));
```

If the interpreter supports only integer values, she might write an aspect in List. 4.3. However, since the interpreter will be extended to support other data types, List. 4.3 is not appropriate. She has to write an aspect in List. 4.2.

Next, another programmer extends the original interpreter to support character strings. Again, he does not have to modify the existing classes. This extension is implemented by the StringAspect in List. 4.4. Since the operator + now represents concatenation of character strings, the around advice implements the new behavior of the eval() method.

We can easily build an interpreter supporting both integers and character strings. It can be composed just by compiling the classes and aspects provided by the two programmers. However, these two aspects conflict with each other, *i.e.* multiple advices are woven at the same join point. Despite this conflict, those advices works together and they do not cause interference since they are designed in advance to be composable by *linearization*. Since AspectJ provides proceed() calls, those advices are connected by proceed() to make a single chain. The interpreter can deal with the AST e constructed by this code:

```
aspect StringAspect {
  Object around(Plus t):
      target(t) && execution(Object Plus.eval()) {
    Object lvalue = t.getLeft().eval();
    Object rvalue = t.getRight().eval();
    if (lvalue instanceof String || rvalue instanceof String) {
      String lstr = lvalue.toString();
      String rstr = rvalue.toString();
      return lstr + rstr;  // not for composition
    } else {
      return proceed(t);
}}}
```

List. 4.4: An aspect for character strings

```
aspect StringAspect {
  Object around(Plus t):
      target(t) && execution(Object Plus.eval()) {
    return (String)t.getLeft() + (String)t.getRight();
}}
```

List. 4.5: The composition-unaware StringAspect aspect

```
  Expression e = new Plus(
    new Constant("Str"), new Constant("1"));
```

Suppose that IntegerAspect is executed first when e.eval() is called. Since the both of the given operands are not Integer and it cannot handle the operands, it calls proceed() to invoke the next advice. This code can be regarded as an AspectJ version of the chain of responsibility pattern. Then StringAspect returns a character string "Str1".

However, these advices are not satisfactory from the software engineering viewpoint. The programmers of each aspect need global reasoning for implementing the aspect to be composable; they must be aware of composition with other (maybe unknown yet) advices. Programmers have to design a composition protocol for the advice chain before implementing each advice body. The protocol design is not easy since the advices must be able to correctly work with and without other advices.

The composition of advices is a crosscutting concern. Note that most statements in List. 4.2 and 4.4 are for the composition by linearization. Only the two return statements marked by a comment implement the behavior of the eval() method in the Plus class. IntegerAspect in List. 4.3 and StringAspect in List. 4.5 are ideal since they are not aware of their composition. However, if we compile these aspects together, we cannot obtain the expected behavior; the advices cause advice interference. The compiler will throws a

```
aspect IntegerStringAspect {
  Object resolver plusEvalIntStr(Plus t)
        and(IntegerAspect.plusEvalInt(t),
            StringAspect.plusEvalStr) {
    Object lvalue = t.getLeft().eval();
    Object rvalue = t.getRight().eval();
    if (lvalue instanceof String && rvalue instanceof String) {
      return [StringAspect.plusEvalStr].proceed(t);
    } else if (lvalue instanceof Integer &&
               rvalue instanceof Integer) {
      return [IntegerAspect.plusEvalInt].proceed(t);
    } else {
      assert lvalue instanceof String ||
      rvalue instanceof String;
      return lvalue.toString() + rvalue.toString();
}}}
```

List. 4.6: An aspect for combining IntegerAspect and StringAspect

ClassCastException evaluating the following ASTs:

```
Expression e = new Plus(
  new Constant("Hello "), new Constant("world!"));
```

or return a character string "12" although both operands are integer values:

```
Expression e2 = new Plus(new Constant(1), new Constant(2));
```

4.2 A New Mechanism for Advice Composition

To resolve the problem mentioned above, we propose a novel language extension of AspectJ. This new language named *Airia* allows programmers to separately describe the composed behavior of advices. The behavior is used as specialized one only when those advices conflict with each other. The original advices are not used unless the resolver explicitly invokes them. We propose resolving conflicts by describing the composed behavior. The composed behavior is separately described in a new kind of around advice called a *resolver*. It is not described in conflicting advices. Hence the implementation of each conflicting advice is independent of the other conflicting advices and their composition protocol.

List. 4.6 is an example of an aspect including a resolver. It resolves the advice interference presented in the previous section. Note that the resolver controls the composition of the conflicting advices by proceed() calls (details of this resolver are mentioned later). Since the code for resolving interference is separated into this resolver, the other advices are not aware of composition.

```
aspect Evaluation {
  public Object Expression.eval() {
    return null;
  }
}

aspect IntegerAspect {
  Object around plusEvalInt(Plus t):
      target(t) && execution(Object Plus.eval()) {
    return (Integer)t.getLeft().eval() +
           (Integer)t.getRight().eval();
  }
}

aspect StringAspect {
  Object around plusEvalStr(Plus t):
      target(t) && execution(Object Plus.eval()) {
    return (String)t.getLeft().eval() +
           (String)t.getRight().eval();
  }
}
```

List. 4.7: Aspects written in our language

See List. 4.7, which presents three advices written in our language. They are simpler than the equivalent aspects shown in List. 4.2 and 4.4. They are almost the same as the ideal aspects in List. 4.3 and 4.5 except that every advice has a unique name. The resolver users these advice names.

Since a resolver is just a new kind of advice, it is composable like normal advice. A conflict caused by that composition can be resolved like a conflict between normal advices. Programmers can resolve that conflict by giving another resolver. The resolver implements the composed behavior used when that conflict happens. Suppose that we write a new aspect CachedEvaluationAspect and its advice conflicts with the advices in IntegerAspect and StringAspect. For resolving conflict among these three advices, we can write a new resolver. Since we already have the resolver in IntegerStringAspect, which resolves the conflict between IntegerAspect and StringAspect, the new resolver can provide the composed behavior of the three advices by reusing the resolver of IntegerStringAspect. It will be described as if it resolves a conflict between the resolver in IntegerStringAspect and the advice in CachedEvaluationAspect.

4.2.1 A Resolver

In Airia, a resolver is a special around advice, which is declared with a keyword resolver instead of around. The syntax of resolver declaration is the following:

```
RetrunType resolver ResolverName(ArgumentType ArgumentName, ...)
  and|or(ConflictingAdviceName[(BoundArgumentName, ...)], ...)
  [uses HelperAdviceName, ...]  { Body }
```

The resolver keyword is followed by a resolver name. A parameter list to the resolver follows the resolver name if any. Unlike normal advices in AspectJ, it does not take a pointcut but it takes an and/or clause, which specifies a list of potentially conflicting advices. The resolver is expected to resolve conflict among these advices. Except the resolver keyword, its name, and the and/or clause, a resolver is the same as an around advice. The return type of a resolver is Object if the join points bound to the resolver have different return types. The body of the resolver may include a proceed() call.

In List. 4.6, a resolver is named plusEvalIntStr and takes an and clause, which lists the names of the around advices in the two aspects IntegerAspect and StringAspect. Note that an advice in Airia also has a unique name. See List. 4.7. The IntegerAspect aspect has an advice named plusEvalInt and the StringAspect has an advice named plusEvalStr. IntegerAspect.plusEvalInt and StringAspect.plusEvalStr are their fully-qualified names.

A resolver is executed at the join points specified by an and/or clause. Since the resolver in List. 4.6 has an and clause, it is executed at the join points that all the given advices are bound to, that is, when the eval() method in the Plus class is executed. Note that those advices of the two aspects IntegerAspect and StringAspect conflict at those join points. A resolver has higher precedence than the advices specified by its and/or clause. Hence, it *overrides* all the conflicting advices at the join points. In our example, when the eval() method in the Plus class is called, the body of the resolver is executed first.

The advices given to the and/or clause of a resolver work as pointcuts. Thus, a resolver can take parameters and pass them to those advices. For example, the resolver in List. 4.6 takes a parameter t and passes it to the advice in the IntegerAspect. The parameter t is bound to the value that this advice binds its parameter to, that is, the target object of the call to the eval() method.

A resolver may have an or clause. This specifies that the resolver is executed at the join points that at least one advice given to the or clause is bound to. For example, the next resolver is executed at the join points that either advice A, B, or C is bound to:

```
Object resolver precedence() or(A, B, C) {
  return [A, B, C].proceed();
}
```

The or clause can be used for specifying precedence order among advices as we do with declare precedence in AspectJ. The resolver shown above specifies that the precedence order is A, B, and C. [A, B, C].proceed() executes the three advices in that order (we below mention this proceed() call again).

We introduced an or clause for reducing the number of necessary resolvers. If we could not use an or clause, we would have to define a number of resolvers for all possible combinations of potentially conflicting advices. Suppose that we have three advices A, B, and C. We would have to define resolvers for every combination: A and B, B and C, C and A, and all the three, if only two of them conflict at some join points (and the other advice is not bound to those join points). Since we expect that resolvers for those combination would share the same body, using an or clause would reduce the number of resolvers we must describe.

Join points selected by an and/or clause are the intersection/union of the join point *shadow* [60] selected for the advices given to that and/or clause, respectively. Dynamic pointcuts such as cflow and target are ignored. Thus, a resolver may be executed at the join points that the advices in its and/or clause are not bound to.

We adopted this language design since it is extremely difficult to detect conflicts among advices even at runtime. Since an advice in AspectJ can change the dynamic contexts, after its body is executed, an advice with a lower precedence than that advice may be removed from the set of the executable advices at that point. Suppose that the pointcut of an advice includes if (Expr.flag) and the value of Expr.flag is true. If an advice with higher precedence than that advice sets Expr.flag to false before calling proceed(), the advice with if (Expr.flag) will not be executed by the proceed() call.

4.2.2 A Proceed Call with Precedence

To implement composed behavior for resolving a conflict, a resolver can call proceed() to invoke another advice with the next highest precedence. The proceed() call from a resolver must explicitly specify the precedence order of the advices given to the and/or clause, which will be invoked by the proceed() call. The precedence order is described between brackets preceding .proceed.

Precedence order is given to each invocation by proceed(). Different precedence order can be given to different proceed() calls in the same resolver. Suppose that there are two advices A and B and they conflict at the join point selected by a pointcut pc(). We assume that there is no other advices. Then resolver AorB can call proceed() twice with different precedence order:

```
void resolver AorB() or(A, B) {
  [A, B].proceed();
  [B, A].proceed();
}

pointcut pc(): ...;

void around A(): pc() {
  proceed();
}

void around B(): pc() {
  proceed();
}
```

Except for giving precedence, the semantics of proceed() call in a resolver is the same as original AspectJ. When [A, B].proceed() is called, A is invoked. B is invoked by a proceed() call in A. A proceed() call in B executes the original computation at the join point. On the other hand, when [B, A].proceed() is called, B is invoked. A is the next. Note that [A, B].proceed() does not mean that executing A and then B. It means that A has higher precedence than B; A or B may not be executed when their pointcuts do not match the current join point.

A proceed() call can remove advices from the set of the remaining advices, which will be invoked by later proceed() calls. If the advice list between brackets does not include an advice given to the and/or clause, that advice is removed. In List. 4.6, both proceed() calls remove one advice. The former removes IntegerAspect.plusEvalInt() and the other removes StringAspect.plusEvalStr(). For example, [IntegerAspect.plusEvalInt].proceed() invokes the plusEvalInt() advice in IntegerAspect and then, if a proceed() is called later, the original eval() method is invoked. The plusEvalStr() advice in StringAspect is never invoked.

4.2.3 Composability of Resolvers

Since a resolver is a special around advice, conflicts among resolvers and normal advices can be resolved by other resolvers. Hence a resolver is composable like normal advice. An and/or clause may include a resolver. A proceed() call with precedence can specify precedence order among resolvers.

Let us consider a new advice shown in List. 4.8. The join point of this advice is the execution of the eval() method. Thus, this advice conflicts

```
aspect CachedEvaluationAspect {
  Object Expression.cachedValue;
  boolean Expression.isChanged = false;
  void around plusEvalCache(Expression t):
      execution(Object Plus.eval()) && target(t) {
    if (t.isChanged) {
      cachedValue = proceed(t);
      t.isChanged = false;
    }
    return cachedValue;
  }

  after changed(Expression t): ... {
    t.isChanged = true;
}}
```

List. 4.8: The EvaluationCacheAspect aspect

```
aspect IntegerStringCacheAspect {
  Object resolver evalIntStrCache():
      and(IntegerStringAspect.evalIntStr,
          EvaluationCacheAspect.plusEvalCache)

    return [EvaluationCacheAspect.plusEvalCache,
            IntegerStringAspect.evalIntStr].proceed();
}}
```

List. 4.9: A resolver resolving conflicts between a normal advice and another resolver

with the two advices in IntegerAspect and StringAspect shown in List. 4.7. Since the conflict between these two advices has been already resolved by the resolver in IntegerStringAspect, we reuse this resolver to describe the composed behavior of these two advices and the new advice. See List. 4.9. This resolver in IntegerStringCacheAspect has an and clause, which lists the new advice in CachedEvaluationAspect and the resolver in IntegerStringAspect. It resolves conflicts between these advice and the resolver.

The behavior of a resolver resolving conflicts among other resolvers (and advices) is the same as that of resolvers resolving conflicts among normal advices. When the eval() method is called, the resolver in IntegerString-CacheAspect is invoked first since it has higher precedence than the other advices and resolver. When this resolver calls proceed() with precedence, the advice with the next highest precedence is executed, which is the advice in CachedEvaluationAspect. Then, if the advice calls proceed(), the resolver in IntegerStringAspect is executed. Note that this resolver does not explicitly describe how the conflicts between IntegerAspect and StringAspect are resolved. It is encapsulated in the resolver of IntegerStringAspect. The composition by IntegerStringCacheAspect is hierarchical.

Existing resolvers can be removed when it cannot be reused. To implement a new resolver that must change the precedence order given by another resolver, programmers can explicitly remove the existing resolver. For example, if a new resolver requires the resolver in IntegerStringAspect should have higher precedence than the advice in CachedEvaluation, the resolver in IntegerStringCacheAspect is removed as if a resolver removes a normal advice. The removed resolver is not executed; the new resolver can define a new precedence order different from the order declared by the removed resolver.

In Airia, advices and resolvers are linearized according to the precedence order specified by proceed() calls. Airia does not provide language constructs similar to declare precedence in AspectJ. The reason we do not use declare precedence is partially composable; if we have the next code:

```
declare precedence:  A, B;
declare precedence:  B, C;
```

then the compiler automatically computes precedence order, which is that A precedes B and B precedes C. However, declare precedence cannot remove other declare precedece statements. On the other hand, a resolver can remove existing resolvers when they are not needed. Furthermore resolvers can flexibly specify different precedence order to a different control-flow path by a proceed() call with precedence. The precedence order in Airia must be explicitly specified. There is no default precedence order unlike in AspectJ.

4.2.4 A Compile Time Check of Conflict Resolution

Our language Airia requires that all conflicts among advices should be explicitly resolved by resolvers. Our compiler checks this requirement at compile time. If programmers declare inconsistent precedence or forget to specify precedence among advices, then our compiler will report errors.

The compiler checks this requirement conservatively. Since the checks is statically done at compile time, the compiler recognized that two advices conflict with each other if they share the same join point shadow (hence they may not be executed at the same join points). Programmers must provide resolvers for all combinations of advices sharing the join point shadow. Otherwise, an error is reported.

A conflict among advices/resolvers is resolved if the resolvers with the highest precedence is uniquely determined at a given join point and at every proceed() call. In Airia, only resolvers and proceed() calls with precedence

Construct	Precedence relations
Type resolver R() and/or(A, B, C)	R $\prec$ A, R $\prec$ B, R $\prec$ C
[A, B, C].proceed()	A $\prec$ B, B $\prec$ C

Table 4.1: Precedence relations declared by constructs in Airia

declare precedence order (Table 4.1). An and/or clause declares that the resolver has higher precedence than the advices or resolvers listed in it. proceed() calls also declare the precedence order among the advices/resolvers between the brackets. Let X $\prec$ Y represent that X has higher precedence than Y. This relation is transitive, *i.e.*, if X $\prec$ Y and Y $\prec$ Z then X $\prec$ Z. Suppose there are three advices A, B, C and they share the same join point shadow and thus conflict with each other. Assume that there are no other advices conflict with them. Then suppose that we describe the following resolvers:

```
void resolver R() and(S, T, A) {
  [S, T, A].proceed();
}
void resolver S() and(A, B) {
  [A, B].proceed();
}
void resolver T() and(B, C) {
  [B, C].proceed();
}
```

They resolve the conflict among A, B, and C. First, R is executed at the join point (shadow) since R has the highest precedence, which is specified by the and clauses of the three resolvers. proceed() calls are ignored at this time. When R calls proceed(), S is executed since it has the highest precedence among the remaining advices and resolvers. Note that we cannot determine S is the highest by the and clauses of the resolvers but now the proceed() call in R declares that S $\prec$ T $\prec$ A. Then, when S calls proceed(), T is executed. Although the proceed() call in S does not declare the precedence of T, we can determine that the next is T because the proceed() call in R declares T $\prec$ A. The proceed() call in T invokes not B but A since the proceed() call in S declares A $\prec$ B and the proceed() call in T declares B $\prec$ C.

If the definition of the resolver R were the following:

```
void resolver R() and(S, T) {
  [S, T].proceed();
}
```

Then the conflict among A, B, and C would not be resolved. When S calls proceed(), no resolver with the next highest precedence is determined since precedence order between A and T is not given.

The precedence order declared by an and/or clause cannot be removed. Even if S is removed by another resolver in the example above, S $\prec$ A and S $\prec$ B declared by the and clause of S are not removed; they are effective. Without this rule, the check of conflict resolution would be extremely complicated. The following resolvers would be valid:

```
void resolver U() and(V, D) {
  [D].proceed(); //remove U
}
void resolver V() and(U, D) {
  [D].proceed(); //remove T
}
```

Their and clauses declare U $\prec$ V and V $\prec$ U and thus the precedence order seems to have a cycle. However, if we first pick up U, since U removes V, the result would be only U $\prec$ V and U $\prec$ D, which has no cycle. On the other hand, if we first pick up V, since V removes U, the result would be different precedence order including no cycle. We introduce the rule to avoid this complication and ambiguity.

Some resolver's body may include multiple proceed() calls declaring different precedence order. The advice invoked at a proceed() call is determined by using only the declarations by that proceed() call and the chain of proceed() calls executed so far until the current join point. Our compiler checks conflict resolution along every conservatively possible control path.

The compiler statically checks that a program will never report an error when advices are executed under the semantics mentioned here. Airia executed advices at a given join point (shadow) in the order determined by the following steps. (1) First, Airia collects all the (potentially) conflicting advices and resolvers at the given join point. During this collection, the dynamic part of the pointcut , such as cflow pointcut of an advice is ignored. Then the compiler makes a directed graph $D(V, E)$ where a vertex $v \in V$ represents an advice or a resolver and an edge $u \prec v \in E$ represents that an advice u has higher precedence than another advice v. If a resolver r has an and/or clause listing a, b, ..., then $r \prec a$, $r \prec b$, ... the compiler also makes an empty set M, whose elements are vertices $v \in V$.

(2) Airia finds a top advice t in D, where $t \prec v$ for any $v \in V \setminus M$. Here, $V \setminus M$ represents a set of the elements contained in V but not in M. If $V \setminus M$ is empty, there is no other advice. Airia executes the original computation at the join point. If there is no unique t or if the graph D has a cycle, an error is reported. Otherwise, Airia executes the body of the top advice t.

```
aspect TraceLogging {
  before log(): ... {
    Logger.getInst().debug(thisJoinPointStaticPart.toString());
}}

aspect ArgumentLogging {
  before log(): ... {
    Object[] args = thisJoinPoint.getArgs();
    Logger.getInst().debug("Args: " + Arrays.toString(args));
}}
```

List. 4.10: Aspects for two kinds of logging

(3) When a proceed() call is encountered during the execution of the body of t, Airia adds t into M. If the proceed() call has precedence order [a, b, c, ...], Airia adds $t \prec a$, $a \prec b$, $b \prec c$, ... to the graph D. If an advice v is included in the and/or clause of t but v is not included in the list of the precedence order of the proceed() call, then v is added to M. Then Airia goes back to Step (2) to execute a top advice t.

(4) If a proceed() call is not encountered, the execution at the join point finishes.

4.2.5 A Helper Advice

To implement composed behavior, a resolver can introduce a new normal advice and give it arbitrary precedence. It usually gives intermediate precedence so that the new advice is executed between the advices the resolver resolves the conflict among.

Suppose that we have two logging aspects shown in List. 4.10. The advice in the TraceLogging aspect records executed methods during program execution. The ArgumentLogging aspect records the values of arguments when a method is invoked. If the precedence order is that TraceLogging is executed before ArgumentLogging, then a printed method name is followed by argument values. However, if a program is multi-threaded, the two advices must be atomically executed. Otherwise, printed log messages will be interleaved as the following:

```
[DEBUG] execution(Object Main.run(String))
[DEBUG] execution(void Test.test())
[DEBUG] Args: []
[DEBUG] Args: [--debug]
```

Here, the fourth line shows the value of the argument to the run() method.

```
aspect LoggingWithSync {
  before lock() {
    Logger.getInst().lock();  //reentrant lock
  }

  before unlock() {
    Logger.getInst().unlock();
  }

  void resolver sync()
      and(TraceLogging.log, ArgumentLogging.log)
      uses lock, unlock {
    [lock, TraceLogging.log, ArgumentLogging.log,
     unlock].proceed();
}}
```

List. 4.11: A resolver for atomically executing two aspects

List. 4.11 shows a resolver for atomic execution of the two logging advices. This resolver uses two helper advices lock() and unlock(). Note that this resolver has a uses clause, which specifies the helper advices for that resolver. The pointcut of a helper advice is not explicitly specified; a helper advice is bound to the same join points that the resolver using that helper advice is bound to. The helper advices can be included in the precedence order of proceed(). In List. 4.11, the proceed() call gives the lock() advice the highest precedence while it gives the unlock() advice the lowest precedence among the four advices. Thus, the lock() advice acquires a lock, the logging advices print messages, and then the unlock() releases before the method logged by the aspects is executed. Without these helper advices, the resolver could not implement atomic execution. Since a lock has to be released after logging advices but before the original computation at the join point, introducing helper advices is the only option. For example, the following resolver does not work:

```
void resolver sync()
    and(TraceLogging.log, ArgumentLogging.log) {
  Logger.getInst().lock();
  [TraceLogging.log, ArgumentLogging.log].proceed();
  Logger.getInst().unlock();
}
```

Multiple resolvers may use the same helper advice. If those resolvers are bound to the same join points, that helper advice is executed only once at every join point (shadow). If a resolver removes another resolver using a helper advice, that helper advice is not removed together. It must be explicitly removed.

4.3 Case Studies

To evaluate Airia, we have rewritten advice conflicts used for case studies in the existing literature. We also compare the existing AOP approaches with regard to capabilities to implement advice composition separately. The advice conflicts used here are listed in Table 4.3 (1–9), which includes ones introduced in this dissertation. Table 4.4 describes if the existing languages and systems can implement advice composition for the advice conflicts (1–9).

As shown in the first row on Table 4.4, Airia provides sufficient expressibility for those advice composition overall although it shows limitations in the advice conflicts that require resolution techniques called conditional execution and/or control of after advices. The results of other languages or systems also tend to depend on resolution techniques required for the conflicts. In the rest of this section, we will discuss each of the results in detail.

Authorization and logging

Interaction between an authorization aspect and a logging aspect is a well-known case that precedence order among aspects must be specified carefully. The authorization aspect prevents execution of methods if the methods are called by actions from users who do not have permission for those actions. List. 4.12 (a) shows an implementation of the authorization aspect. It contains an around advice executed before such methods and checks if the user has permission to execute them; only when she has permission, the aspect executes the protected method by calling proceed(). Next, the logging aspect records that specified method has been executed. An implementation of it is shown in List. 4.12 (b). Its before advice just writes a user name given to the parameter and the method name to be executed.

The composed behavior of those aspects is quite different depending on precedence order between them since the authorization aspect has a control-flow modification [3]. Suppose that the precedence of the authorization aspect is higher than that of logging aspect, the logging aspect only makes a log message only when the authorization aspect allows executing the method. If programmers intended that they would like to detect potential attacks on the system from the log messages, this composed behavior is not expected; they must give higher precedence to the logging aspect.

All the languages and systems are sufficient for the composition of those aspects since the composition simply gives precedence order between the aspects. In Airia, programmers can implement the composition with a resolver as shown in List. 4.12. It only gives higher precedence to the logging aspect

	Composition strategy
AO	Advice-level ordering
CC	Context-aware composition
CE	Conditional execution
ME	Mutual execution
IC	Implicit cut
RM	Results merging

Table 4.2: List of the composition strategies

	Advice conflicts	Used composition strategies
1	Authorization and logging [3, 66]	
2	Answering machine and call-forwarding [25]	CC
3	Compression and encryption [57, 58, 59]	AO
4	Connection pooling and synchronization [5]	IC
5	Integer and string features	CC, ME, IC
6	Raise checker and persistence [63]	CE
7	Trace and arguments logging	IC
8	Two kinds of toString [71, 28]	RM
9	Two kinds of filters for web applications	CE

Table 4.3: List of the advice conflicts and the composition strategies used for each conflict

	Feature interactions								
Languages or systems	1	2	3	4	5	6	7	8	9
Airia	y	y	y	y/n	y	n	y	y	y-
AspectJ	y	n	n	y-/n	n	n	y-	n	n
Declarative aspect composition	y	n	n	y-/n	n	y	y-	n	y
Context-aware composition rules	y	y	y	y-/n	n	n	y-	n	n
OARTA	y	n	y	y/y-	y	y-	y-	n	y-
POPART	y	y	y	y-/n	y-	n	y-	n	n
Reflex	y	n	y	y/n	n	n	y	n	n
Stateful aspect	y	y-	y	y-/n	n	n	y-	n	n

y: possible; **y-**: possible with limitations; **n**: impossible

Table 4.4: Comparison of the languages/systems w.r.t. the capabilities for implementing the composition separately

```
aspect Authorization {
  void around auth(String user):
      execution(void Server.doAction(..)) && args(user) {
    if (isAllowed(user)) {
      proceed(user);
    } else {
      System.out.println("not permitted");
    }
  }
}
```

(a) Authorization.aj

```
public aspect Logging {
  before log(String user):
      execution(void Server.doAction(..)) && args(user) {
    System.out.println(
      user + " is trying to execute doAction()");
  }
}
```

(b) Logging.aj

```
aspect AuthLogging {
  void resolver authLog() and(Authorization.auth, Logging.log) {
    [Logging.log, Authorization.auth].proceed();
  }
}
```

(c) AuthLogging.aj

List. 4.12: The aspects for authorization and logging and the resolver for the composition of those aspects by using Airia's proceed() call.

Answering machine and call-forwarding

The feature interaction between an answer machine feature and a call-forwarding feature in a phone management system is a classic example in the telecommunication domain. Dinkelaker et al. implement those features by using aspects for a case study of their meta-aspect protocol, POPART [25]. We have rewritten their aspect into AspectJ (Airia) as shown in List. 4.13 (a) and (b). The ToAnswerMachine aspect extends the Phone.receiveCall() method so that it can pass the call to the answering machine if nobody receives that call. The return value of the receiveCall() method indicates whether someone received the call or not. Calling the recieveCall() method on an answering machine object will start up the answering machine. On the other hand, the ForwardCall aspect forwards a call to the another phone registered in advance if nobody received the call. It also extends the receiveCall() method of the Phone class and calls the receiveCall() method on the forward phone if the original methods returns false.

To implement the composition between those aspects, languages or systems need to support a composition strategy called *context-aware composition*, which changes precedence order between conflicting advices or removes some of the advices depending on the dynamic context. When both of these features are enabled, their appropriate composed behavior is that the call-forwarding feature does not forward a call if the answering machine of the forward phone is active. This is because an original callee cannot hear any recorded messages if they are recorded by the answering machine of the forward phone.

The AnswerMachineAndCallForwarding aspect in List. 4.13 (c) implements that composition in Airia. The resolver of AnswerMachineAndCallForwarding aspect invokes the advice of ToAnserMachine by the first proceed() call if an answer machine is active for a forward phone. While the other languages and systems supporting context-aware composition including POPART can implement this composition as well, AspectJ does not support it. AspectJ's declare precedence statically specifies precedence order between aspects.

Compression and encryption

As mentioned in Section 2.5.2.4, there are advices programmers must specify their precedence order at not an aspect level but an advice level. List. 4.14 shows the Compression aspect and the Encryption aspect, which requires

```
aspect ToAnswerMachine {
  boolean around checkAndReceive(Phone t, String phoneNumber):
    execution(boolean Phone.receiveCall(String)) && this(t) &&
    args(phoneNumber)
  {
    boolean answered = proceed(t, phoneNumber);
    AnswerMachine am = t.getAnswerMachine();
    if (!answered && am.isActive()) {
      answered = am.receiveCall(phoneNumber);
    }
    return answered;
  }
}
```

(a) The ToAnswerMachine aspect

```
aspect CallForwarding {
  boolean around forward(Phone t, String phoneNumber):
    execution(boolean Phone.receiveCall(String)) && this(t) &&
    args(phoneNumber)
  {
    boolean answered = proceed(t, phoneNumber);
    Phone to = t.getForwardPhone();
    if (!answered && to != null) {
      answered = to.receiveCall(phoneNumber);
    }
    return answered;
  }
}
```

(b) The CallForwarding aspect

```
aspect AnswerMachineAndCallForwarding {
  boolean resolver receiveCall(Phone t, String s)
      and(ToAnswerMachine.checkAndReceive,
          CallForwarding.forward(t, s)) {
    Phone to = t.getForwardPhone();
    if (to != null && to.getAnswerMachine().isActive()) {
      return [ToAnswerMachine.checkAndReceive,
              CallForwarding.forward].proceed(t, s);
    } else {
      return [CallForwarding.forward,
              CallForwarding.forward].proceed(t, s);
    }
  }
}
```

(c) The AnswerMachineAndCallForwarding aspect

List. 4.13: The aspects for authorization and logging and the resolver for the composition of those aspects *advice-level ordering.* These aspects are applied to an application that reads data from a file, process that data, and then writes the output to a file. The Compression aspect enables to save disk space by compressing data before they are save to file. It also decompresses the compressed data before they are used in the application. The Encryption aspect writes encrypted data to a file and allows that application to read the encrypted data. Although the Compression.compressData() advice must be executed before the Encryption.encryptData() advice, Encryption.decryptData() must be executed before Compression.decompressData().

Airia supports advice-level ordering and hence can implement such composition by the resolvers in List. 4.14 (c). Programmers can specify precedence order between those advices by using their unique name in a proceed() call.

Note that if decompressData() and decryptData() were after advices and their pointcuts were:

```
execution(void Application.load()) && target(a)
```

then programmers would not need advice-level ordering for those aspects. Since an after advice with lower precedence is executed before ones with higher precedence, they can obtain the same behavior by:

```
declare precedence: Compression, Encryption;
```

Connection pooling and synchronization

The next example is from an SPL for peer-to-peer overlay networks [5], which was originally implemented by aspect refinements in Aspectual Mixin Layers (AML) [10]. AML is a language and composition system that integrates aspects into refinements; namely, it allows programmers to extend definitions of existing advices by refinements. List. 4.15 shows an aspect implementing an object-pooling feature for network connection class. When communication session is closed, the aspect prevents ClientConnection objects from being deleted so that the object can be reused for the next session. Another feature of the SPL is to add synchronization to the aspects for that pooling feature. In AML, programmers can implement such synchronization by refining the Pooling aspect with the code below:

```
aspect Compression {
  before compressData(Application a):
      execution(void Application.save()) && target(a) {
    zip(a.data);
  }

  before decompressData(Application a):
      execution(void Application.processData()) && target(a) {
    unzip(a.data);
  }
}
```

(a) Compression.aj

```
aspect Encryption {
  String key = "...";

  before encryptData(Application a):
      execution(void Application.save()) && target(a) {
    encrypt(a.data, key);
  }

  before decryptData(Application a):
      execution(void Application.processData()) && target(a) {
    decrypt(a.data, key);
  }
}
```

(b) Encryption.aj

```
aspect CompressionEncryption {
  void resolver save()
      and(Encryption.encryptData, Compression.compressData) {
    [Compression.compressData, Encryption.encryptData].proceed();
  }

  void resolver load()
      and(Encryption.decryptData, Compression.decompressData) {
    [Compression.decompressData,
     Encryption.decryptData].proceed();
  }
}
```

(c) The resolver for Compression and Encryption

```
Application app = new Application();

app.load();
app.processData();
app.save();
```

(d) The order of invocation of methods defined in Application

List. 4.14: The Compression aspect and the Encryption aspect

```
refines aspect Pooling {
  ClientConnection getFromPool(ClientSocket sock) {
    synchronized(pool) {
      return super.getFromPool(sock);
    }
  }
}
```

This refinement replaces the getFromPool() advice of the Pooling aspect in List. 4.15 with a new advice defined in the refinements. In the refinement, super.getPool() will invoke the original advice in the synchronized block.

A refinement of an around advice can be achieved by a normal advice executed at the same join points with the refined advice. To pick join points when specified advice is to be executed, some languages and systems provide a mechanism called *implicit cut* [74]. In Airia, and/or clauses can be regarded as mechanisms for implicit cuts. The syncGetFromPool() resolver in List. 4.15 (b) implements the refinement above. The and clause executes that resolver at the same join points with the original advice, and then the resolver invokes the original by proceed(). Note that even though programmers can implement such an advice without an implicit cut by giving the same pointcut to the original advice, the pointcut of the advice is fragile; they must update that pointcut when the pointcuts of the original advice is changed.

However, implicit cuts cannot emulate refinements of after advices. In this example, the putToPool() advice must be synchronized as well. The next method refinement:

```
ClientConnection putToPool(ClientSocket sock) {
  synchronized(pool) {
    return super.putToPool(sock);
  }
}
```

synchronizes only execution of putToPool. The syncPutToPool() resolver in List. 4.15 (b) is a workaround in Airia. The resolver executes not only the putToPool() advice but also executes original computation at the join point wrongly in the synchronized block. Reflex's implicit cuts similarly cannot emulate refinements of after advices. OARTA, in contrast, provides pointcuts picking out execution of a specified advice, adviceexecution(). It allows rewriting the refinement of putToPool() into an around advice with adviceexecution(Pooling.putToPool).

```
aspect Pooling {
  HashMap pool = new HashMap();

  ClientConnection around getFromPool(String s):
      call(ClientConnection.new(..)) && args(s) {
    if (!pool.containsKey(s)) {
      return proceed(s);
    } else {
      return (ClientConnection)pool.get(s);
    }
  }

  after putToPool(ClientConnection t):
      execution(void ClientConnection.close()) && target(t) {
    this.pool.put(t.getServer(), t);
  }
}
```

(a) The Pooling aspect

```
aspect Sync {
  Object resolver syncGetFromPool() and(Pooling.getFromPool) {
    synchronized(Pooling.aspectOf().pool) {
      Object res = [Pooling.getFromPool].proceed();
      return res;
    }
  }

  // workaround for a refinement of after advice
  Object resolver syncPutToPool() and(Pooling.putToPool) {
    synchronized(Pooling.aspectOf().pool) {
      Object res = [Pooling.putToPool].proceed();
      return res;
    }
  }
}
```

(b) The Synchronization aspect

List. 4.15: Aspects for class pooling and its synchronization.

Integer and string features

In Section 4.2, we have already explained how programmers can combine the IntegerAspect and StringAspect by a resolver. Here, we review what kinds of composition strategies are necessary for that combination of aspects. First, the composition requires *mutual exclusion* [51]. Mutual exclusion is a strategy that executes exactly one advice from a combination of advices. In our example, when the Plus.eval() method is executed, the resolver selects IntegerAspect or StringAspect to invoke according to the actual types of the operands. Next, it also uses context-aware composition to select an aspect from the two aspects according to the actual type of operands.

In the aspect-oriented languages or systems listed above, only Reflex provides mechanism directly supporting mutual exclusion. However, it does not support dynamic composition, and hence it cannot implement composition of those aspects. On the other hand, OARTA can emulate mutual exclusion by appending if pointcuts to the conflicting advices. In order to achieve mutual exclusion of IntegerAspect and StringAspect, programmers append:

```
&& if (t.getLeft().eval() instanceOf Integer &&
       t.getRight().eval() instanceOf Integer)
```

to the advice of IntegerAspect by using OARTA's andpointcut. They also append a pointcut that replaces Integer with String in the pointcut above to the advice of StringAspect. Since context-aware composition rules and POPART can remove unnecessary conflicting advices, they can implement conditional execution.

Finally, the composition needs implicit cuts to implement addition of an integer value and a character string. OARTA does not have a mechanism for selecting shared join points. Context-aware composition rules and POPART also do not support implicit cuts and hence cannot implement that extra code at the conflicting join points without copying pointcut definitions.

Raise checker and persistence

The paper [63] proposed its unique composition strategy named *conditional execution* as a part of *declarative aspect composition*. The strategy defines composed behavior of two conflicting aspects; each advice returns a Boolean value, and only when the first advice returns true, the other advice can be executed. List. 4.16 shows two after advices, which should be executed in conditional execution. They are part of a personal management system used in companies. The first advice is executed when the salary of an employee is increased, and checks if the updated salary is not higher than her manager. If

```
public aspect CheckRaise {
  after(Employee t, int level):
      execution(void Employee.increaseSalary(..)) &&
      args(level) && target(t) {
    Employee m = t.getManager();
    if (m != null && t.getSalary() + level >= m.getSalary()) {
      return true;  // used for declarative aspect composition
    } else {
      t.salary += level; // revert salary
      return false; // used for declarative aspect composition
    }
  }
}
```

(a) The CheckRaise aspect

```
aspect EmployeePersistence {
  after(Employee e):
      execution(void Employee.increaseSalary(..)) && target(e) {
    // save to DB
  }
}
```

(b) The EmployeePrsistence aspect

List. 4.16: Aspects for a personal management system (pseudo code in AspectJ-like syntax)

the raise is acceptable, it returns true. The other advice is for persistence. It saves states of employee objects into a database. By specifying those advices are executed in conditional execution, the persistence aspect is executed only when the salary of an employee has been successfully raised.

Resolvers in Airia cannot achieve such composition since it has no way to control after advice depending on the results of other advices that have been executed before. OARTA can emulate conditional execution by keeping the result of previous raise-checking into a field. Programmers can prevent the persistence advice from being executed by appending if pointcut that ensures the raise was accepted.

Trace and arguments logging

We have introduced the trace-logging aspect and argument-logging aspect in Section 4.2.5. The languages or systems other with implicit cuts are sufficient for the composition of those logging advices. If programmers tolerate fragility of pointcuts, all the languages or systems can achieve that composition; programmers add new advices for synchronization and specify precedence order between the advices and the original logging advices.

Two kinds of toString

Let us consider a figure editor application on which users draw shapes such as rectangles and circles. List. 4.17 (a) and (b) introduce two aspects, the Color aspect and the Border aspect, for the figure editor. The Color aspect let shape objects have their colors by an inter-type declaration. It also extends the toString() method of the Shape class so that the result of that method can include information about colors of objects. The other aspect, Border adds border properties to shape objects and extends the Shape.toString() method for the same purpose with Color.

Some readers might feel the two around advices extending the toString() method are strange since they do not have proceed() call to invoke the original toString() method. This example was originally used to show how traits allow method composition without dispersal of glue code as described in Section 2.5.2.2. We have rewritten normal toString() methods of traits to around advices.

Airia can *merge* the results of conflicting around advices in a resolver. The ColorAndBorder.toStreing() resolver (List. 4.17 (c)) implements composition of the around advices defined in Color and Border. The resolver executes the original method by removing those advices with proceed() call. It next executes the Color.toString() advice and the Boder.toString() advice separately. Finally, the resolver concatenates the return values of the original methods and those advices for the result of the toString() method. The other languages and systems on Table 4.4 cannot handle the return values of conflicting advices.

Two kinds of filters for web applications

The last example is two kinds of authorization aspects, which are used for a web application developed by Java Servlet. The AopHttpServlet class in List. 4.18 (a) is a base class of the application. A dynamic web page is implemented by a subclass of this class as shown in List. 4.18 (b). The subclass is called a servlet, and it has a method invoked when the corresponding page is requested by a web browser. This method generates a page described in HTML and writes it to ServletOutputStream, which is sent through a network and displayed on the web browser.

Here, we use aspects to implement a filter-like mechanism, which perform an extra action before contents are generated while J2EE provides an application-programming interface (API) for such mechanism. The first aspect is the ErrorHandling aspect shown in List. 4.18 (c). This aspect returns error messages to a user when a servlet encounters an error while generating

```
aspect Color {
  int Shape.color = 0x000000;

  String around toString(Shape t):
      execution(String Shape+.toString()) && target(t) {
    Formatter f = new Formatter();
    f.format("color: #%06X", t.color);
    return f.toString();
  }
}
```

(a) The Color aspect

```
aspect Border {
  int Shape.borderWidth = 1;

  String around toString(Shape t):
      execution(String Shape+.toString()) && target(t) {
    return "border: " + t.borderWidth;
  }
}
```

(b) The ColorAndBorder aspect

```
aspect ColorAndBorder {
  String resolver toString()
      and(Color.toString, Border.toString) {
    String org = [].proceed();
    String color = [Color.toString].proceed();
    String border = [Border.toString].proceed();
    return org + " [" + color + "; " + border + "]";
  }
}
```

(c) The ColorAndBorder aspect implementing composition of Color and Border

List. 4.17: Composition of two toString() in a figure editor

```
abstract class AopHttpServlet extends HttpServlet {}
```

(a) A base class for servlets

```
class Index extends AopHttpServlet {
  protected void service(HttpServletRequest req,
                         HttpServletResponse res)
      throws ServletException, IOException {
    ServletOutputStream out = res.getOutputStream();
    out.print(/* generated content */);
}}
```

(b) A class implementing a dynamic web page

```
aspect ErrorHandler {
  void around doFilter(HttpServletRequest req,
                       HttpServletResponse res):
      execution(AopHttpServlet+.service(..)) && args(req, res) {
    try {
      proceed(req, res);
    } catch(Exception e) {
      // generate error messages
}}}
```

(c) The ErrorHandling aspect

List. 4.18: The classes of a Java Servlet application

a page.

List. 4.19 shows aspects for two kinds of authorization. First, the SessionAuth aspect prevents a user from accessing a page unless the user is already logged in with her password. The advice is executed before a page is generated. If a user does not logged in, it throws AccessDeniedException; the ErrorHandling aspect will catch the exception and display a message. Otherwise, it calls proceed() and runs the original service() method implemented by each servlet. On the other hand, the HostAuth aspect permits accesses only from a trusted network in the same way with SessionAuth; it also throws an AccessDeniedException if the access is not permitted.

Programmers can easily implements the conjunction of the two policies; the application is accessible to users who are in a trusted network and have logged in. The following resolver defines precedence order between those aspects so that after the SessionAuth allows a user to access, HostAuth checks where access is from:

```
void resolver auth()
    and(SessionAuth.doFilter, HostAuth.doFilter) {
  [SessionAuth.doFilter, HostAuth.doFilter].proceed();
}
```

However, it is not straightforward to implement the disjunction of the two authentication policies. Suppose that we allow accesses from trusted

```
aspect SessionAuth {
  void around doFilter(HttpServletRequest req):
      execution(void AopHttpServlet+.service(..)) &&
      args(req, HttpServletResponse) {
    if (req.getRemoteUser() == null) {
      throw new AccessDeniedException("...");
    }
    proceed(req);
  }
}
.................................................................
aspect HostAuth {
  void around doFilter(HttpServletRequest req):
      execution(void servlet.AopHttpServlet+.service(..)) &&
      args(req, HttpServletResponse) {
    if (!isPermitted(req.getRemoteAddr())) {
      throw new AccessDeniedException("...");
    }
    proceed(req);
  }

  boolean isPermitted(String remoteAddr) {
    return remoteAddr.startsWith("192.168.0.");
  }
}
.................................................................
aspect SessionOrHostAuth {
  void resolver doFilter()
      and(SessionAuth.doFilter, HostAuth.doFilter) {
    try {
      [HostAuth.doFilter].proceed();
    } catch (AccessDeniedException e) {
      [SessionAuth.doFilter].proceed();
    }
  }
}
```

List. 4.19: Two authorization aspects

```
aspect SessionOrHostAuth {
  void resolver doFilter()
      and(SessionAuth.doFilter, HostAuth.doFilter) {
    try {
      return [SessionAuth.doFilter].proceed();
    } catch(AccessDeniedException e) {
      return [HostAuth.doFilter].proceed();
}}}
```

(a) The SessionOrHostAuth aspect

```
aspect AuthErrorHandler {
  void resolver doFilter()
      or(SessionOrHostAuth.doFilter, ErrorHandler.doFilter) {
    [ErrorHandler.doFilter,
     SessionOrHostAuth.doFilter].proceed();
}}
```

(b) The AuthErrorHandler aspect

List. 4.20: Composition of the two authorization aspect

network *or* a user who has already logged in. Implementing such composed behavior needs more complex resolvers as shown in List. 4.20 (a) than the conjunction. After one filter rejects an access, the resolver retries the other filter by using exception handler. It executes the advice of HostAuth first. If the advice denies an access, it throws an exception without executing the original method. The exception will be caught by the resolver, which invoke the advice of SessionAuth from its catch block. The AuthErrorHandler aspect in List. 4.19 (b) is used to give higher precedence to the ErrorHandler aspect than the authorization aspects. Note that the composition between the authorization aspects can be considered as conditional execution since SessionAuth can be executed only after HostAuth failed. Airia can emulate conditional execution of around advice although it cannot emulate that of before advice.

4.4 Implementation

We have implemented an Airia compiler by extending an AspectJ compiler named the AspectBench compiler [13][1]. Several languages based on AspectJ have used the AspectBench compiler to implement their compilers. Its front-end is implemented on the JastAddJ Java compiler by using JastAdd compiler-compiler framework as GluonJ and FeatureGluonJ are. On

[1]The source code of Airia is available from:
http://www.csg.ci.i.u-tokyo.ac.jp/projects/airia/

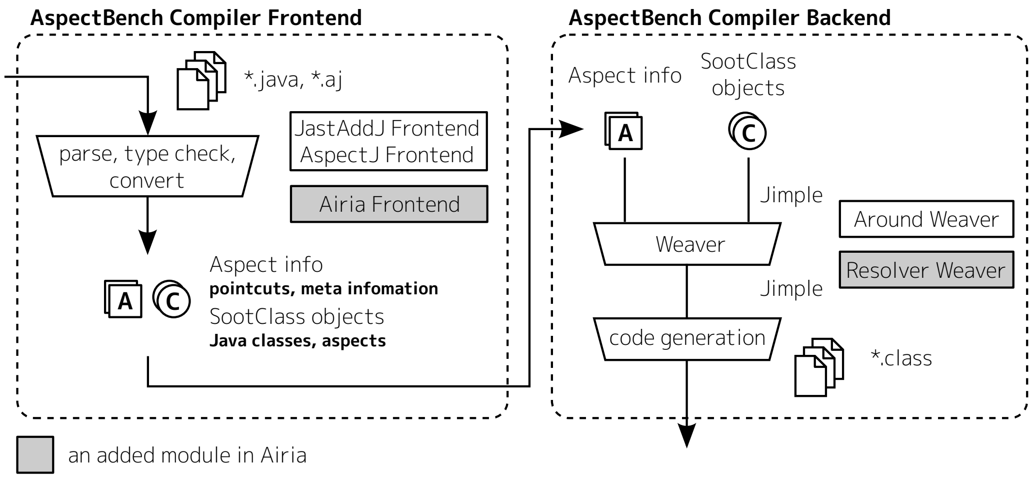

Fig. 4.1: An overview of Airia implementation

the other hand, its back-end uses Soot framework [77] to weave aspects into classes. Airia's weaver modifies and translates the code into Jimple intermediate representation, which is simplified bytecode used in Soot, to insert code for invoking advices.

Figure 4.1 illustrates an overview of the Airia compiler. Since the front-end is sufficient extensible, we just added additional syntax rules and aspects unique to Airia. The front-end of the AspectBench compiler translates aspects into classes and meta information called *aspect info*, which retains advice types and pointcuts. We have introduced a new kind of aspect info for representing resolvers. After the classes are converted to SootClass (*i.e.*, meta class provided by Soot) objects, the front-end forwards the objects and aspect info to its back-end.

To support resolvers by extending the back-end of the AspectBench compiler, it was necessary to duplicate and modify the existing code for weaving aspects. Although the procedure for weaving a single resolver is almost the same as a normal around advice, we have reimplemented the process for weaving multiple aspects at the same join points. The compiler makes a tree starting from a resolver with the highest precedence ending with a node representing original computation at the join point shadow; nodes and edges on the tree represent advices and proceed() calls. It weaves advices for every path from the root to a leaf recursively. While it handles a resolver with multiple proceed() calls, it weaves other advices to be executed by one of the proceed() call. Then it moves the woven code including the advices and the original computation into a closure method. After that, it weaves advices for other proceed() call. Finally, the proceed() calls are replaced with invocations of the closure methods. In our implementation of the compiler, the size of generated code could be slightly large. It is possible to reduce the size if multiple paths contain the same sequences of advices.

4.5 Summary

We presented a language extension of AspectJ. This language named Airia can resolve interference among conflicting advices, which could not be satisfactorily resolved in original AspectJ. Airia provides a resolver, which is a language construct to enable programmers to implement composed behavior separately from conflicting advices by resolvers. In a resolver, an existing advice can be invoked by `proceed()` call with precedence. Unnecessary advices can be also removed by that call. Airia thereby provides better composability to advices than AspectJ. Since a resolver is a new kind of advice, it is also composable by other resolvers. We have implemented an Airia compiler by extending the AspectBench compiler using JastAdd. This compiler checks that all conflicts are resolved by resolvers and consistency of precedence order declared by resolvers and `proceed()` calls.

Chapter 5

Concluding Remarks

Feature interaction is a major cause that degrades composability in software product line development. This dissertation addressed feature interaction at two different levels. FeatureGluonJ enables to reduce the number of derivatives between two generalization groups of features. We found higher-order redundancy in the derivatives of MobileMedia. In FeatureGluonJ, such derivatives can be implemented as a generic feature module, which is written by using the super feature modules of those groups as an interface.

On the other hand, the latter half of this dissertation addressed feature interaction at the lower level. When multiple features are used together, method extension mechanisms contained in their feature modules may conflict and cause interference. The method extension mechanisms include advices in AspectJ and methods of revisers in GluonJ. To address this problem in AspectJ, we have developed its language extension named Airia. It provides a new kind of advice called a resolver, which implements composed behavior of conflicting advices. To implement it, resolvers can execute only specified conflicting advices by `proceed()` call with precedence and merge the result of the advices.

Contributions

The contribution of this dissertation is as the following:

- We have developed a new language FeatureGluonJ supporting hierarchical implementation of features. Although it is not novel as shown in Section 2.4.7, our contribution is designing language semantics that are necessary and sufficient expressibility for implementing product lines by reducing complexity of virtual classes and virtual revisers.

- This dissertation has shown that inheritance of feature modules is useful to reduce the number of redundant derivatives. A super feature module defines an interface that specifies virtual classes and revisers provided by its sub-features. The abstraction by the interface allows programmers to write a generic feature module applicable to derivatives between groups of sub-features.

- This dissertation also proposes a new language, Airia. Its unique language construct, resolver, allows programmers composition of conflicting advices whose interference are not avoidable by naive linearization. A resolver can implement composed behavior of conflicting advices; it invokes only specified advices by removing unnecessary ones with Airia's `proceed()` call with precedence and merges the results of the invocations. Airia also supports advice-level ordering; the `proceed()` call controls the execution of conflicting advices at the advice level.

- We have shown that composability of composition code is also important and designed a construct composable by itself. Since a resolver is a new kind of advice, it is also composable by other resolvers.

Future Directions

Integration of resolvers with FeatureGluonJ

Since multiple revisers may extends the same method, conflict of method extensions is common to FeatureGluonJ. Integrating resolver with FeatureGluonJ should be useful to resolve that conflict. However, one issue with this integration is that giving precedence among all conflicting advices explicitly is annoying even though the order of conflicting advices is commutative.

Formalization of language semantics

Both FeatureGluonJ and Airia does not have formal definitions of semantics. FeatureGluonJ is based on GluonJ and light-weight family polymorphism which have formal definitions to prove they are mostly modular and type safe, respectively. The formal definition of FeatureGluonJ will be valuable to show that it derives those properties.

Bibliography

[1] Autoconf—gnu project—free software foundation (fsf). `http://www.gnu.org/software/autoconf/`.

[2] Automake—gnu project—free software foundation (fsf). `http://www.gnu.org/software/automake/`.

[3] M. Aksit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 39–50. ACM, 2009.

[4] J. Aldrich. Open modules: Modular reasoning about advice. In A. Black, editor, *ECOOP 2005 — Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2005.

[5] S. Apel and C. Kästner. Aspect refinement — unifying AOP and stepwise refinement. *Journal of Object Technology*, 6(9):13–33, 2007.

[6] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[7] S. Apel, C. Kastner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[8] S. Apel, T. Leich, and G. Saake. Aspect refinement and bounding quantification in incremental designs. In *Asia-Pacific Software Engineering*

Conference, pages 796–804, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[9] S. Apel, T. Leich, and G. Saake. Mixin-based aspect inheritance. In *Technical Report Number 10*, Germany, 2005. Department of Computer Science, University of Magdeburg.

[10] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 122–131, New York, NY, USA, 2006. ACM.

[11] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022–1047, Nov. 2010.

[12] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Berlin Heidelberg, 2006.

[13] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.

[14] E. W. Axelsen and S. Krogdahl. Adaptable generic programming with required type specifications and package templates. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 83–94, New York, NY, USA, 2012. ACM.

[15] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] D. Batory, P. Höfner, and J. Kim. Feature interactions, products, and composition. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 13–22, New York, NY, USA, 2011. ACM.

[17] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, Oct. 1992.

[18] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, 2004.

[19] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: controlling the scope of change in java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 177–189, New York, NY, USA, 2005. ACM.

[20] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*, pages 59–62, 1989.

[21] S. Chiba. Load-time structural reflection in java. In E. Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Berlin Heidelberg, 2000.

[22] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular composition of crosscutting structures by contextual predicate dispatch. Technical Report C-267, Dept. of Math. and Comp. Sciences, Tokyo Institute of Technology, 2009.

[23] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 539–554, New York, NY, USA, 2010. ACM.

[24] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *SIGPLAN Not.*, 34(10):325–339, Oct. 1999.

[25] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 51–62, New York, NY, USA, 2009. ACM.

[26] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM*

SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering, GPCE '02, pages 173–188, London, UK, UK, 2002. Springer-Verlag.

[27] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 141–150, New York, NY, USA, 2004. ACM.

[28] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, Mar. 2006.

[29] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[30] E. Ernst. Propagating class and method combination. In R. Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 67–91. Springer Berlin Heidelberg, 1999.

[31] E. Ernst. Family polymorphism. In J. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer Berlin Heidelberg, 2001.

[32] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM.

[33] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '08, pages 261–270. ACM, 2008.

[34] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, RIACS, 2000.

[35] V. Gasiunas and I. Aracic. Dungeon: A case study of feature-oriented programming with virtual classes. In *Proceedings of the 2nd Workshop on Aspect-Oriented Product Line Engineering*, 2007.

[36] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 133–152, New York, NY, USA, 2007. ACM.

[37] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer Berlin Heidelberg, 2003.

[38] S. Herrmann, C. Hundt, and M. Mosconi. OT/J language definition v1.3. `http://www.objectteams.org/def/1.3/s9.html#s9.3`, May 2011.

[39] C. Hundt, K. Mehner, C. Preiffer, and D. Sokenou. Improving alignment of crosscutting features with code in product line engineering. *Journal of Object Technology*, 6(9):417–436, 2007.

[40] A. Igarashi and M. Viroli. Variant path types for scalable extensibility. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 113–132, New York, NY, USA, 2007. ACM.

[41] T. Kamina and T. Tamai. Lightweight dependent classes. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 113–124, New York, NY, USA, 2008. ACM.

[42] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SE-90-TR-021, Carnegie Mellon University, 1990.

[43] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, Jan. 1998.

[44] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 223–232. IEEE Computer Society, 2007.

[45] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

[46] C. Kästner, S. Apel, and K. Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, volume 2 of *SPLC '11*, pages 5:1–5:8. ACM, 2011.

[47] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, July 2012.

[48] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 181–190. Carnegie Mellon University, 2009.

[49] E. Katz and S. Katz. Semantic aspect interactions and possibly shared join points. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2010, pages 43–51. School of Electrical Engineering and Computer Science, University of Central Florida, 2010.

[50] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001.

[51] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering*, volume 2177 of *Lecture Notes in Computer Science*, pages 57–69. Springer Berlin / Heidelberg, 2001.

[52] S. Krogdahl, B. Moller-Pedersen, and F. Sorensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.

[53] J. Lauret, H. Waeselynck, and J.-C. Fabre. Detection of interferences in aspect-oriented programs using executable assertions. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pages 165–170, Washington, DC, USA, 2012. IEEE Computer Society.

[54] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM.

[55] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, pages 68–77, New York, NY, USA, 2006. ACM.

[56] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Proceedings of Object-oriented programming, systems, languages, and applications*, OOPSLA, pages 397–406. ACM, 1989.

[57] A. Marot and R. Wuyts. Composability of aspects. In *SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, pages 1–6, New York, NY, USA, 2008. ACM.

[58] A. Marot and R. Wuyts. A DSL to declare aspect execution order. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–5, New York, NY, USA, 2008. ACM.

[59] A. Marot and R. Wuyts. Composing aspects with aspects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 157–168, New York, NY, USA, 2010. ACM.

[60] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2002, pages 17–26, 2002.

[61] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 90–99. ACM, 2003.

[62] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 127–136, New York, NY, USA, 2004. ACM.

[63] I. Nagy, L. Bergmans, and M. Aksit. Declarative aspect composition. In *Software-engineering Properties of Languages for Aspect Technologies, SPLAT!*, AOSD 2004, 2004.

[64] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 39–50, New York, NY, USA, 2006. ACM.

[65] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 235–250, New York, NY, USA, 1995. ACM.

[66] R. Pawlak, L. Duchien, and L. Seinturier. CompAr: Ensuring safe around advice composition. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin Heidelberg, 2005.

[67] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin Heidelberg, 1997.

[68] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18:285–331, 2008.

[69] F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[70] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2010.

[71] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *ECOOP 2003 — Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Berlin Heidelberg, 2003.

[72] S. Schulze, S. Apel, and C. Kästner. Code clones in feature-oriented software product lines. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 103–112, New York, NY, USA, 2010. ACM.

[73] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 21–29, New York, NY, USA, 2003. ACM.

[74] É. Tanter. Aspects of composition in the Reflex AOP kernel. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 2006.

[75] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[76] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012.

[77] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot — a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–23. IBM Press, 1999.

[78] T. Young and G. Murphy. Using AspectJ to build a product line for mobile devices. AOSD Demo., 2005.