An Advice for Advice Composition in AspectJ

Fuminobu Takeyama and Shigeru Chiba

Tokyo Institute of Technology, Japan
http://www.csg.is.titech.ac.jp/~{f_takeyama, chiba}

Abstract. Aspect composition often involves advice interference and this is a crucial problem in aspect oriented programming. When multiple advices are woven at the same join point, the advices often interfere with each other. Giving appropriate precedence order is a typical solution of this problem but it cannot resolve all kinds of advice interference. To address this problem, we propose a novel language extension named *Airia*, which provides a new kind of around advice for resolving advice interference. This kind of advice named a *resolver* is invoked only at the join points when given advices conflict with each other. The resolvers can call an extended version of proceed, which takes as an argument precedence order among remaining advices. Furthermore, the resolvers are composable. They can be used to resolve interference among other resolvers and advices.

1 Introduction

In aspect-oriented programming, crosscutting concerns are modularized into aspects. Composing a new aspect by combining existing aspects is not easy. The aspects often conflict with each other and cause undesirable interference. This problem is called *aspect interference*. In particular, resolving interference among advices in the aspects is a serious issue that has been investigated in the research community. This paper addresses this issue called *advice interference*.

A typical solution is to allow programmers to control the precedence order among conflicting advices. For example, AspectJ [10] provides declare precedence for this control. However, for some combinations of advices, there is no correct precedence order with which the composed behavior is acceptable. Such combination needs modifying the bodies of the conflicting advices to explicitly implement the merged behavior. This is not desirable since the programmers have to be aware of the composition when they write an individual advice. The implementation of the composition should be described separately from the conflicting advices.

This paper proposes a novel extension of AspectJ, which is named *Airia*. In this language, a new kind of around advice called *resolvers* are available. A resolver is used to implement the composition of conflicting advices. It is invoked at the join points when the given set of advices conflict with each other. Since a resolver has higher precedence than those conflicting advices, it overrides the implementation of those advices. In the body of the resolver, a proceed call takes

```
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;
}}
```

Listing 1. Classes representing the AST

as an argument the precedence order among those conflicting advices. It can thereby control the execution order of the remaining advices. In our language Airia, therefore, declare precedence is not available. Furthermore, a resolver is composable. It can implement the composition of other resolvers and advices. Our language Airia has been implemented by using the AspectBench compiler [4] and JastAdd [8] [1]. In summary, the contribution of this paper is to propose a new language construct named a *resolver*. The advantages are two. (1) It enables separation of the implementation of advice composition while keeping sufficient expressive power. Also, (2) it is composable and thus we can implement composition of several advices in a hierarchical manner.

In the rest of this paper, Section 2 shows a motivating example. Section 3 presents the design of a resolver. Section 4 describes implementation of Airia compiler. Section 5 mentions related work and Section 6 concludes this paper.

2 A Motivating Example

We first show an example of aspect interference that is not resolved in existing approaches.

2.1 A Simple Interpreter

We present a simple interpreter with a binary operator +, which is written in AspectJ. Listing 1 shows classes representing AST (Abstract Syntax Tree)

[1] The source code of Airia is available from:
`http://www.csg.is.titech.ac.jp/`

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

Listing 2. An aspect for integer values

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

Listing 3. An aspect for character strings

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.

We then extend the interpreter to support integer values. We do not have to modify the existing classes. We have only to write a new aspect shown in Listing 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. When e.eval is executed on this tree, it returns 3.

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

Next, we extend the original interpreter to support character strings. Again, we do not have to modify the existing classes. We implement this extension by the StringAspect in Listing 3. Since the operator + now represents concatenation of character strings, the around advice implements the behavior of the eval method. If we compile the classes and the aspects in Listing 1 and 3, then the resulting interpreter will correctly handle character strings.

The last step is to build an interpreter supporting both integers and character strings. Some readers might expect that we could easily obtain the interpreter if the two aspects IntegerAspect and StringAspect are compiled and woven together. However, these two aspects conflict with each other, *i.e.* multiple advices are woven at the same join point. The resulting behavior of the eval method is different from our naive expectation. This paper deals with this unexpected behavior of the combined advices, which we call *advice interference*. It is one kind of *aspect* interference; other kinds of aspect interference such as [9] are out of the scope of this paper.

If the two advices are combined, the resulting eval method cannot process all acceptable ASTs. In AspectJ, when the eval method is called, the advice with the highest precedence is executed. Suppose that IntegerAspect has the highest.

```
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) {
            return (Integer)lvalue + (Integer)rvalue; // not for composition
        } else {
            return proceed(t);
}}}

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) {
            return lvalue.toString() + rvalue.toString(); // not for composition
        } else {
            return proceed(t);
}}}
```

Listing 4. A composable version of the conflicting advices

Then the eval method does not process the AST e constructed by the following code:

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

It will throw ClassCastException since the operands are character strings but the around advice in IntegerAspect assumes that the operands are integer values. Changing the precedence order does not solve this problem. In AspectJ, we can explicitly specify precedence. For example,

```
declare precedence: StringAspect, IntegerAspect;
```

this declaration specifies that StringAspect has higher precedence than IntegerAspect. The eval method now returns an unexpected value when the AST e2 shown below is evaluated:

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

The returned value will be a character string "12" although both operands are integer values.

2.2 An Incomplete Solution in AspectJ

A partial solution of the problem above is to make advices composable by *linearization*. Since AspectJ provides proceed calls, we can reimplement advices and connect them by proceed to make a single chain. If proceed is called in an advice body, it invokes the advice with the next highest precedence in the chain. If there is no other advice, the original computation at the join point is executed.

Listing 4 shows the result of the reimplementation to use the linearization. The resulting code can be regarded as an AspectJ version of the chain of responsibility pattern. Now the two around advices call proceed to invoke the next

```
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 {
            throw new RuntimeException();
}}}
```

Listing 5. The aspect for combining IntegerAspect and StringAspect

advice when they cannot deal with the given operands. Although the combination of some advices requires an explicit declaration of the precedence order for linearization, the two aspects in Listing 4 do not require it; they correctly work under any precedence order. The interpreter containing these two aspects can deal with the AST constructed by this code:

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

However, this solution is not satisfactory from the software engineering viewpoint. Programmers need global reasoning for combining advices; they must be aware of other (maybe unknown yet) advices. Furthermore the implementation of each advice body includes the composition code for linearization, which connects it to other 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 is a crosscutting concern. Note that most statements in Listing 4 are for the composition by the linearization. Only the two return statements marked by a comment implement the behavior of the eval method in the Plus class. The composition code scatters over the two advices.

3 A New Approach for Resolving Interference

To resolve the problem mentioned above, we propose a novel language extension of AspectJ. This new language named *Airia* allows programmers to separately describe how to resolve advice interference. Instead of the conflicting advices themselves, the resolving code is described in a new kind of around advice called a *resolver*. Hence the implementation of each conflicting advice is independent of the other conflicting advices and their composition protocol.

Listing 5 is an example of an aspect including a resolver. It resolves the advice interference presented in the previous section. Details of this resolver are mentioned below. Since the resolving code is separated into this resolver, the conflicting advices do not include the code for composition or resolution of the interference. See Listing 6, which presents the three conflicting advices written

```
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 t.getLeft().eval().toString() + t.getRight().eval().toString();
}}
```

Listing 6. The aspects written in our language

in our language. They are simpler than the composable version of the aspects shown in Listing 4. They are the same as the original aspects in Listing 2 and 3 except that every advice has a unique name. These advice names are used by the resolver.

A resolver is composable. Programmers can write a resolver that resolves interference among other resolvers and normal advices. Suppose that we write a new aspect EvaluationCacheAspect and its advice causes interference with the advices of IntegerAspect and StringAspect. For these three conflicting advices, we can write a new resolver by reusing the existing resolver of IntegerStringAspect. Since the resolver of IntegerStringAspect deals with the advice interference between IntegerAspect and StringAspect, the new resolver will be declared to deal with the interference between the resolver of IntegerStringAspect and the new advice of EvaluationCacheAspect. The implementation of that new resolver will call proceed to execute the resolver of IntegerStringAspect.

3.1 A Resolver

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 interference 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 Listing 5, 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 also has a unique name. See Listing 6. 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.

The join points when a resolver is executed are specified by an and/or clause. Since the resolver in Listing 5 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 Listing 5 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 only the two advices A and B are bound to but C is not:

```
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 many 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 they conflict at different join points. Since we expect that those combinations would share the same body, using an or clause would reduce the number of resolvers we must describe.

To be precise, the join points selected by an and/or clause are the intersection/union of the join point *shadow* [14] 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.

3.2 A Proceed Call with Precedence

Like a normal advice, a resolver can call proceed to invoke another advice with the next highest precedence. The proceed call from a resolver explicitly specifies the precedence order of the advices given to the and/or clause, which will be invoked by the proceed call. Note that unlike AspectJ our language Airia does not provide declare precedence. The precedence order is described between brackets preceding .proceed.

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 a 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();
}
```

When [A, B].proceed() is called, A is invoked. B is invoked by the proceed call in A. The 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 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.

The 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, the advice is removed. In Listing 5, 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 it calls proceed again, the original eval method is invoked. The plusEvalStr advice in StringAspect is never invoked.

```
aspect EvaluationCacheAspect {
    Object Expression.cachedValue;
    boolean Expression.isChanged = false;
    void around plusEvalCache(Expression t): execution(Object Plus.eval()) && args(t) {
        if (t.isChanged) {
            cachedValue = proceed(t);
            isChanged = false;
        }
        return cachedValue;
    }
    after changed(): ... {
        isChanged = true;
}}
```

Listing 7. The EvaluationCacheAspect aspect

```
aspect IntegerStringCacheAspect {
    Object resolver evalIntStrCache():
            and(IntegerStringAspect.evalIntStr, EvaluationCacheAspect.plusEvalCache)
        return [EvaluationCacheAspect.plusEvalCache,
                IntegerStringAspect.evalIntStr].proceed();
}}
```

Listing 8. A resolver resolving conflicts between a normal advice and another resolver

3.3 Composability of Resolvers

A resolver, which is a special around advice, may also conflict with other resolvers or normal advices. This conflict can be also resolved by another resolver; a resolver is composable. An advice given to an and/or clause may be a resolver. A proceed call with precedence specifies precedence order among conflicting advices and/or resolvers.

Let us consider a new advice shown in Listing 7. The join point of this advice is the execution of the eval method. Thus, this advice conflicts with the two advices in IntegerAspect and StringAspect shown in Listing 2 and 3. Since the conflict between these two advices has been already resolved by the resolver in IntegerStringAspect, we reuse this resolver to resolve the conflicts among the new advice and these two advices. See Listing 8. This resolver in IntegerStringCacheAspect has an and clause, which lists the new advice in EvaluationCacheAspect and the resolver in IntegerStringAspect. It resolves conflicts between the advice and the resolver.

The behavior of a resolver for another resolver is the same as normal resolvers. When the eval method is called, this resolver in IntegerStringCacheAspect 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 EvaluationCacheAspect. After that if the advice calls proceed, the resolver in IntegerStringCacheAspect is executed. Note that this resolver does not explicitly describe how the conflicts between IntegerAspect and StringAspect are resolved. It is encapsulated in the

Table 1. A summary of precedence relations declared by constructs in Airia

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

resolver of IntegerStringAspect. The composition of IntegerStringCacheAspect is hierarchical.

Existing resolvers can be overridden when it cannot be reused. To implement a new resolver that changes the precedence order given by another resolver, programmers explicitly remove the other resolver. For example, if a new resolver requires the resolver in `IntegerStringAspect` should have higher precedence than the advice in EvaluationCache, the resolver in `IntegerStringCacheAspect` is removed by the same way that a resolver removes a normal advice. The removed resolver is not executed; the new resolver can define a new precedence order among the advices that were resolved by the removed one.

Unlike declare precedence in AspectJ, a resolver can flexibly modify precedence order among conflicting advices even during runtime by a proceed call with precedence. Thus, declare precedence is not available in our language Airia. The precedence order must be explicitly specified; there is no default precedence order unlike AspectJ.

3.4 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.

To enable statically checking whether conflicts are resolved or not, our definition of conflict is conservative like AspectJ. If advices partly share their join point shadow, they conflict. Due to our specification mentioned in Section 3.1, a resolver cannot have dynamic residue. Thus our compiler can statically determine conflicting advices at every join point shadow and the resolvers executed at that shadow.

Our compiler recognizes that a conflict has been resolved if the advice or resolver with the next highest precedence is always determinable. Also the highest resolver executed first at the join point must be uniquely determined. Recall that, in our language Airia, constructs that declare precedence order are a resolver and a proceed call with precedence as summarized in Table 1. The and/or clause declares that the resolver has higher precedence than the advices or resolvers specified in it. Then the proceed call declares precedence order as specified between its brackets. This declarations are effective only in the remaining chain of advices. Here we use a binary relation; $X \prec Y$ represents 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. It must be total order; otherwise, it causes an error. Suppose three advices A, B, C and the following resolvers:

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

They conflict at the same join point shadow and no other advices nor resolvers exists. The first executed resolver is R because of R $\prec$ S given by the and clause of R. After S is invoked by the proceed call in R, the proceed call in S invokes C. This is because of S $\prec$ C and C $\prec$ A given by the proceed call in R. Thus this conflict is resolved. On the other hand, if we rewrite R as follows, then the conflict is not resolved:

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

The declared relations are only R $\prec$ S, S $\prec$ C, S $\prec$ A, and A $\prec$ B; there is no precedence order between A and C.

The precedence order declared by a 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 are still effective. Without this rule, the check of conflict resolution would be extremely complicated. The following resolvers would be valid:

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

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

Some resolvers include multiple proceed calls declaring different precedence order. The next advice invoked at a proceed call is determined dependently on the chain of proceed calls executed so far at the current join point. Our compiler checks conflict resolution along every conservatively possible control path. Please refer to our companion paper [17] for more detail.

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

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

Listing 9. Two aspects for logging

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

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

Listing 10. A resolver for synchronizing two aspects

3.5 A Helper Advice

A resolver can add a new advice for helping composition. Since a resolver has higher precedence than conflicting advices, the added advice is normally given intermediate precedence among those conflicting advices.

Suppose that we have two logging aspects shown in Listing 9. 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 specifies 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 synchronized. Otherwise, printed log messages will be interleaved as the following:

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

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

Listing 10 shows a resolver for synchronizing 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 are included in the precedence order of proceed. In Listing 10, the lock

```
aspect IntegerStringAspect {
    Object around(Plus t): execution(Object Plus.eval()) && target(t) {
        Object lvalue = t.getLeft().eval();
        Object rvalue = t.getRight().eval();
        if (lvalue instanceof String || rvalue instanceof String) {
            return lvalue.toString() + rvalue.toString();
        } else if (lvalue instanceof Integer && rvalue instanceof Integer) {
            return (Integer)lvalue + (Integer)rvalue;
        } else {
            throw new RuntimeException();
    }}
    declare precedence: IntegerStringAspect, IntegerAspect, StringAspect;
}
```

Listing 11. Another incomplete solution in AspectJ

advice is given the highest precedence while the unlock advice is given 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 synchronization since it had to release a lock between the logging advices and the logged method but the resolver automatically obtains higher precedence than the logging advices.

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.

3.6 Discussion

A resolver does not take a normal pointcut but an and/or clause — a list of conflicting advices. It can call proceed with precedence. These are unique features of our language Airia. To clarify their benefits, we show another aspect in Listing 11. Like the aspect written in Airia, this aspect does not require us to modify the conflicting aspects in Listing 2 and 3. We wrote this aspect in AspectJ to be similar to the aspect written in Airia shown in Listing 5. The aspect has a normal around advice. We manually translated the and clause of the resolver into a normal pointcut for this around advice. In the body of this around advice, we also manually inlined the body of the conflicting advices since a proceed call with precedence is not available.

This aspect has two drawbacks. First, the pointcut of the advice is fragile. We will have to modify the pointcut of this advice when the pointcuts of the conflicting advices are modified. Second, the body of this advice contains code duplication since we manually inlined the body of the conflicting advices. We will also have to modify the advice body when the bodies of the conflicting advices are modified. The aspect written in Airia does not have these problems.

4 Implementation

We have implemented an Airia compiler by extending an AspectJ compiler named the AspectBench compiler (abc). Its front-end is implemented as an extension of the JastAddJ extensible compiler. The extension to abc is implemented by aspects like the example in Section 2.

The current implementation of our compiler does not support the same level of optimization as abc. Unlike abc, our proceed call is implemented only by using closure objects. The size of generated code could be large. A closure is created for each proceed call of every path of advice chain because the selection of invoked advice by the proceed call depends on which proceed calls have been executed at the join point.

5 Related Work

Resolving aspect interference for aspect composition is a classic research topic and hence there have been a number of proposals. For example, Douence *et al.* proposed an approach for detecting and resolving conflicts between aspects on their formal framework, Stateful Aspect [6, 7]. Their approach is making a composition operator extensible so that the operator will generate correctly merged behavior when the operands are conflicting with each other. Although this approach is similar to ours, we provide a single composition operator (*i.e.* a resolver) but we do not make the semantics of the operator extensible. We also propose an extension of AspectJ based on our approach.

Most previous approaches are categorized into meta programming. POPART [5] provides a meta-aspect protocol. Advice composition can be dynamically customized by an instance of MetaAspectManager. Programmers can define an appropriate MetaAspectManager to implement a custom composition policy for resolving conflicts among particular advices. JAsCo [16] also provides a mechanism like this. However, meta programming is often complicated and difficult. Thus, in OARTA [13], the ability for meta programming is restricted. For resolving aspect interference, OARTA allows an advice to modify only the pointcut of another advice. On the other hand, our approach does not need meta programming. A resolver is a special around advice but it is still a base-level language construct.

Context-Aware Composition Rules [11, 12] allows programmers to control precedence order among advices for every join point. It also allows removing an existing advice at some join points. However, as we mentioned in Section 2, some kinds of advice interference cannot be resolved by only reordering advices. On the other hand, our language Airia also provides the ability for adding a new advice only at the join points where advices are conflicting with each other.

Reflex [18] is an infrastructure for building an aspect system. It provides an application-programming interface (API) for implementing a new policy for advice composition. Programmers can exploit this API for customizing the aspect system to resolve conflicts among particular advices. On the other hand, our

language Airia provides a base-level language construct for resolving conflicts. The users of Airia do not have to consider the implementation of the language.

Aspect refinement and mixin-based aspect inheritance [2, 3] enable programmers to incrementally extend the behavior of an existing advice. JastAdd also supports refinement. On the other hand, our language Airia enables extending the behavior of a combination of multiple advices.

The relationship between an around advice and the original computation at the join point is similar to the relationship between a mixin and a class. A `proceed` call corresponds to a `super` call. The former executes the next advice or the original computation while the latter executes the method in the next mixin or class. While advice interference is a problem, interference among mixins given to the same class is also a problem. Traits [15] is a solution for mixin interference. Our approach can be regarded as an application of the idea of traits to aspects. Both approaches allow programmers to define a new advice/method for overriding advices/methods and resolving their conflicts.

6 Conclusion

We presented a language extension of AspectJ. This language named Airia can resolve interference among conflicting advices, which we could not satisfactorily resolve in original AspectJ. Airia enables programmers to separate composition code into an independent resolver, which is a new kind of advice. A resolver is composable. It can resolve interference between another resolver and an advice. We have implemented an Airia compiler by extending the AspectBench compiler using JastAdd.

Our future work includes improving the expressive power of `proceed` calls. There are some proposals such as [1], which are used to detect whether or not advices are commutative, *i.e.* whether or not their combined behavior is independent of their precedence order. In the current design of Airia, programmers have to explicitly specify the precedence order among advices even though they are commutative. This is annoying.

References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 39–50. ACM, New York, NY, USA (2009)
2. Apel, S., Leich, T., Saake, G.: Aspect refinement and bounding quantification in incremental designs. Asia-Pacific Software Engineering Conference 0, 796–804 (2005)
3. Apel, S., Leich, T., Saake, G.: Mixin-based aspect inheritance. In: Technical Report Number 10. Department of Computer Science, University of Magdeburg, Germany (2005)

4. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development. pp. 87–98. ACM, New York, NY, USA (2005)
5. Dinkelaker, T., Mezini, M., Bockisch, C.: The art of the meta-aspect protocol. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 51–62. ACM, New York, NY, USA (2009)
6. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering. pp. 173–188. Springer-Verlag, London, UK (2002)
7. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development. pp. 141–150. ACM, New York, NY, USA (2004)
8. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 1–18. ACM, New York, NY, USA (2007)
9. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. In: AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development. pp. 85–95. ACM, New York, NY, USA (2007)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming. pp. 327–353. Springer-Verlag, London, UK (2001)
11. Marot, A., Wuyts, R.: Composability of aspects. In: SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies. pp. 1–6. ACM, New York, NY, USA (2008)
12. Marot, A., Wuyts, R.: A DSL to declare aspect execution order. In: DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages. pp. 1–5. ACM, New York, NY, USA (2008)
13. Marot, A., Wuyts, R.: Composing aspects with aspects. In: AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development. pp. 157–168. ACM, New York, NY, USA (2010)
14. Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Proc. of Foundations of Aspect-Oriented Languages Workshop. pp. 17–26. AOSD 2002 (2002)
15. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP 2003 – Object-Oriented Programming. pp. 327 – 339 (2003)
16. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development. pp. 21–29. ACM, New York, NY, USA (2003)
17. Takeyama, F.: A new kind of advice for advice composition without interference. Master's thesis, Tokyo Institute of Technology, Japan (2010)
18. Tanter, É.: Aspects of composition in the Reflex AOP kernel. In: Software Composition. Lecture Notes in Computer Science, vol. 4089, pp. 98–113. Springer Berlin / Heidelberg (2006)