Automatic Detection to the Behavioral Conflict in AOP Application Based on Design by Contract

Chengwan He\textsuperscript{1,2}

\textsuperscript{1} Hubei Province Key Laboratory of Intelligent Robot, Wuhan Institute of Technology
\textsuperscript{2} School of Computer Science and Engineering, Wuhan Institute of Technology, Wuhan, China
Email: hechengwan@hotmail.com

Zheng Li
School of Computer Science and Engineering, Wuhan Institute of Technology, Wuhan, China
Email:lzjsj2008@163.com

Abstract—Behavioral conflict is one of the key issues in the practical application of AOP (Aspect-Oriented Programming) technology. Based on the ideas of design by contract and behavioral subtyping, we propose an approach that detects the behavioral conflict automatically at runtime. Use Java annotation to describe the contracts of the base program and aspect code, then extract these contracts through the contract transformation program, and convert them to the assertion verification program, consequently it achieves automatic detection to the behavioral conflict at runtime.

Index Terms—AOP (Aspect-Oriented Programming), behavioral conflict, design by contract, behavioral subtyping, assertion verification

I. INTRODUCTION

Aspect-Oriented Programming \cite{1}\cite{2} approach makes many different concerns independent mutually, such as functional and non-functional requirements of the software system, platform performance and so on, achieving a better modularization. It is considered as a necessary complementarity to the Object-Oriented technology. Generally speaking, the Aspect-Oriented software is composed of two parts: base program to implement the system’s functions and aspect to implement the cross-cutting concerns. An aspect also consists of two parts: pointcut and advice. A pointcut is a set of join points where an advice should be executed. An advice is code that is executed when a join point is reached. AspectJ \cite{3} and Aspectwerkz \cite{4} are AOP languages used widely now.

At present, the Aspect-Oriented technology is well along toward the practicality. However, it faces a key issue in the practical application: behavioral conflict. Behavioral conflict is also known as the semantic conflict of aspect composition \cite{5}\cite{6}. Such conflicts may occur at the following cases: i) Originally, the program is able to run correctly, but it won’t run properly after weaving the aspects; ii) Multiple aspects are woven into a shared join point in different orders, it may give rise to a conflict; iii) It is a mutex relationship between two aspects, which can not be woven into the base program at the same time, and so on. Behavioral conflict of aspect composition may occur between aspect and base program, or between two aspects.

Although the behavioral conflict can be detected, to some extent, in the testing phase of software, testing can detect errors in the program only, however, it can not guarantee that there are no errors existed. That is, software testing cannot detect all of the behavioral conflicts. Moreover, the error locating of the software testing becomes very complicated, due to the separation and encapsulation of the cross-cutting concerns in the AOP system. Therefore, we believe that although software testing is an effective means to guarantee that the program satisfies the user’s requirements, it is not entirely suitable for detection to the behavioral conflicts among aspects.

With the expansion of the software’s scale, as well as the increase in the number of aspects, using manual methods to control the right composition between the aspect and base program produces errors easily, or even impossible. Thus, methods and tools are needed urgently to detect such conflicts automatically.

Design By Contract \cite{7}\cite{8}\cite{9} is used in a very wide range in the field of the Object-Oriented applications, which guarantees the correctness of a method’s behavior by specifying the pre-condition, post-condition and invariants. At the same time, it can accurately specify where the program violates the contracts.

Behavioral Subtyping \cite{10}\cite{11} presents a subtyping relationship, it takes such relationship into consideration not only from structure but from behavior. For example, Class C1 includes the method A, and C2 inherits C1, if the instance of C1 can be replaced by C2’s, thus C2 is called as behavioral subtyping of C1. In other words, assuming that R1 and E1 respectively represent the pre-condition and post-condition of method A in the super Class C1, while R2 and E2 respectively represent the pre-condition and post-condition of method A in the sub-Class C2, so they meet the logical relationship as follows: \((R1 \rightarrow (E2 \rightarrow E1)).\) That is, the pre-condition of the method A in the sub-class C2 becomes weaker, while the post-condition becomes stronger.
This paper presents an approach, based on design by contract and behavioral subtyping, to detect behavioral conflicts. It uses Java annotation [12] to describe the contracts, then extracts these contracts through the contract transformation program, and converts them to the assertion verification program, consequently achieving automatic detection to the behavioral conflict at runtime.

This paper first introduces the behavioral conflict as well as the related work, and then it elaborates the automatic detection approach based on design by contract and behavioral subtyping and its implementation algorithm. Finally, it explains effectiveness of the approach through an example.

II. DESCRIPTION OF THE PROBLEM AND RELATED WORK

A. Description of The Problem

Behavioral conflict is also known as the semantic conflict of aspect composition. It mainly represents that the program is able to run correctly, but it won’t run properly after weaving the aspects. Next, we cite the example (Fig. 1) in reference [6] to illustrate this problem.

![Figure 1. Jukebox system](image1)

Fig. 1 represents a jukebox system, if a song is selected in the JukeboxUI, the method play is called in the Jukebox, then this method calls the play method in Player which is connected to the audio subsystem. Now we add a new requirement to this system which states that check whether the user has enough credits before calling the play method. If there are enough credits, withdraw one credit for calling the play method each time.

class Jukebox {
    ...
    public void play(Song song) {
        Player p = new Player();
        p.play(file);
    }
    ...
}

aspect CreditsAspect {
    before (): call (public void Jukebox.play (Song)) {
        if (Credits.instance ().enoughCredits()) {
            Credits.instance ().withdraw ();
            System.out.println("Ok! Begin playing…");
        } else {
            throw new NotEnoughCreditsException ();
        }
    }
}

![Figure 2. Definition of Jukebox class and CreditsAspect](image2)

In Fig. 2, the left shows the base program while the right shows the aspect code (using AspectJ language). Assuming that the system requires that it should play 10 songs at least before adding this requirement, while currently the user’s credits are less than 10, so it will not be able to satisfy the original requirements of the system after adding this requirement (weaving CreditsAspect), namely, the program can’t run correctly after weaving this aspect. In other words, there is a conflict existed between the CreditsAspect and the base program.

B. Related Work

Reference [13] puts forward an approach based on the behavior subtyping to deal with contract checking (mainly contracts in the methods of a class or interface) after inheriting classes and interfaces in the object-oriented programming. It points out that contract checking tools should report the following three kinds of errors: pre-condition errors, post-condition errors as well as contract inheritance errors. Moreover, it states that, without the last kind of errors, contract checking tools will assign blame incorrectly for some contract violations, and sometimes it even can’t detect certain contract violations.

Reference [14] extends JML (Java Modeling Language), proposes and realizes a kind of contract description language, Pipa, which is suitable for AOP system. Namely, description methods of the pre-conditions, post-conditions and invariants in the advice of aspect. In order to use some support tools of JML, Pipa transforms AOP program and its contract description to Java code and its corresponding contract description. So it doesn’t focus on resolving automatic detection to the behavioral conflict in aspect composition.

Reference [15] applies DBC to AOP, shows corresponding execution sequences of aspects and the assertions (contracts) they should follow, and introduces CONA used to DBC in AOP. It extracts the contracts of aspect, then generates a new aspect through CONA’s processing, to check the contracts. But it is not intended to detect the behavioral conflict in aspect composition.

Reference [5] proposes a model-based method for the conflict detection, it extracts the related information between the Aspect and Class from the UML model in the Aspect-Oriented software, and analyzes the potential conflicts among aspects based on these information. However, it can only detect the conflicts among aspects acted on a shared join point.

Reference [6] proposes a detection model of semantic conflict, based on the composition filter model, which transforms the semantics of filter operations to the resource operations. The needed behaviors (semantics) can be specified by patterns which are used to operate the resources. In order to detect conflicts, the pattern, used to identify the error operations, must be existed. By analyzing all of the advice acted on a shared join point, it detects conflicts based on sequences of the resource operations. However, it needs to add annotations to all advice using the resource-operation specification. And this approach can also only detect the conflicts among aspects acted on a shared join point.

Reference [16] analyzes execution sequences and their dependence of multiple aspects acted on a shared join point, on this basis, identifies a set of requirements upon
mechanisms for composing aspects at a shared join point, and proposes a model based on contracts. The model is used to define contracts specification upon all possible compositions of aspects acted on a shared join point, and detects the possible existed conflicts at runtime. By extending the notion of join points, the proposed model is adopted by AspectJ and Compose*.

Reference [17] proposes a reflective multi-level framework developed for the construction of aspect-oriented applications, which includes a component, the Conflict Manager, which manages and solves the conflicts generated by interaction of multiple aspects at runtime according to the information specified by the application developer. In addition, it introduces a visual tool, Alpheus, which supports the specification of all the components of an application (such as basic objects, aspects etc) as well as the associations and conflicts between the components. Moreover, it can generate the corresponding Java code of the application, and it provides the visualization of different UML diagrams to aid the development process. As a result, aspect-oriented applications are easy to specify and implement.

In summary, behavioral conflict in AOP has aroused widespread attention and produced some research achievements. However, most studies still have some shortcomings, which haven’t solved this problem essentially. For example, it not only needs to solve conflicts which may arise among aspects, but also those may occur between aspects and the base program. In addition, it is necessary to consider that multiple aspects acted on a shared join point as well as acted on different join points.

III. AUTOMATIC DETECTION TO THE BEHAVIORAL CONFLICT BASED ON DESIGN BY CONTRACT AND BEHAVIORAL SUBTYPING

The basic idea of this approach is: apply concepts of design by contract and behavioral subtyping to the Aspect-Oriented software, which uses Java annotation to describe contracts in the base program and aspects, considering the program as super type before weaving a certain aspect, the woven program as a subtype. If they meet the conditions of the behavioral subtyping (pre-condition becomes weaker while post-condition becomes stronger), consequently, we can ensure the correctness of the program’s behaviors after weaving aspects, so as to achieve the automatic detection to the behavioral conflict among aspects.

For the automatic detection to the behavioral conflict in the aspect composition, we use the following flow as shown in Fig. 3. Firstly, according to the inheritance (including Class, Interface, Aspect inheritance) and the weaving relationship between the aspects (including weaving sequence, weaving type), extract the contract description in source code and transform them into the assertion verification program through the contract transformation program, then generate the byte code of Java by the AOP compiler. Consequently automatic detection to the behavioral conflict is achieved at runtime.

A. Description of Contracts

Java annotation is a mechanism to describe the metadata, which is introduced by JDK1.5 and its later versions. We use annotation to describe pre-conditions and post-conditions of the base program and aspect code, and the annotation is defined as shown in Fig. 4.

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface pre {
    String value() default "";
}

@Retention(RetentionPolicy.RUNTIME)
public @interface post {
    String value() default "";
}
```

Figure 4. Definition of the annotation

In Fig. 4, @retention indicates that how long annotations with the annotated type are to be retained. RetentionPolicy.RUNTIME means that annotations are to be recorded in the class file by the compiler and retained by the VM at runtime, so they may be read reflectively. @pre, @post, the declared types of annotation, can be used to describe pre-conditions, post-conditions of methods and aspects, as shown in Fig. 5.

```java
Class Compute{
    @pre("x>=0 ")
    double sqrt(double x){
        ... }
}

aspect before CallCompute {
    @pre ( "a>=0 " )
    @post ( "a==0 " )
    before(double a):call(double Compute.sqrt(double))&&args(a){
        ... }
}
```

Figure 5. Usage of the annotation

Next, we discuss respectively detection to the behavioral conflict after weaving before advice, after advice into the base program.

B. Weaving Before Advice

Fig. 6 describes the base program and aspect code annotated by the contracts.
If considering the program as super type before weaving a certain aspect, the woven program as a subtype, so the relationship between them is shown in Fig. 7.

In order to guarantee correctness of the program’s behavior, the contracts, needed to be satisfied before and after weaving aspects, are shown in Table I. $a_{\text{pre before}}$ denotes that pre-conditions of before advice in aspect code; $a_{\text{post before}}$ denotes that post-conditions of before advice in aspect code; $m_{\text{pre}}$ denotes that pre-conditions of the method m; $m_{\text{post}}$ denotes that post-conditions of the method m; $A \rightarrow B$ denotes that if A then B holds true.

In Table I, the first column lists the contracts which need to be met before weaving aspect into the base program, namely, pre-conditions and post-conditions of the method m; the second column lists the contracts (according to the order) which need to be checked after weaving aspect into the base program. Considering the program as super type before weaving aspect, the woven program as a subtype, when checking these contracts, it requires that the program before and after weaving aspect, satisfies the conditions of behavioral subtyping. Thus, after weaving before advice, pre-conditions of the program become weaker, namely, $m_{\text{pre}} \rightarrow a_{\text{pre before}}$.

### Table I.

<table>
<thead>
<tr>
<th>Contracts before weaving aspect</th>
<th>Contracts after weaving aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_{\text{pre}}$</td>
<td>$a_{\text{pre before}}$</td>
</tr>
<tr>
<td>$m_{\text{post}}$</td>
<td>$a_{\text{post before}}$</td>
</tr>
</tbody>
</table>

After weaving aspect, according to the contract transformation program, convert the above contracts to the assertion verification program as shown in Fig. 8.

C. Weaving After Advice

Weaving after advice is basically similar to weaving before advice. Their differences mainly lie in the contracts to be satisfied after weaving aspect. The base program and aspect code annotated by contracts are shown in the following Fig. 9.

After weaving aspect, according to the contract transformation program, the assertions are converted to the assertion verification program as shown in Fig. 8.

© 2011 ACADEMY PUBLISHER
In order to guarantee that the program executes correctly, the contracts, needed to be satisfied before and after weaving aspects, are shown in Table II. The meaning, indicated by \( m_{\text{pre}} \), \( m_{\text{post}} \), is similar to the above description. \( \text{aft}_{\text{pre}} \) denotes that pre-conditions of after advice in aspect code while \( \text{aft}_{\text{post}} \) denotes that post-conditions of after advice in aspect code.

In Table II, the first column is same to Table I; the second column lists the contracts (according to the order) which need to be checked after weaving aspect (after advice) into the base program. According to the conditions of behavioral subtyping, post-conditions of the program become stronger after weaving after advice, namely, \( m_{\text{pre}} \rightarrow \text{aft}_{\text{post}} \).

### Table II

<table>
<thead>
<tr>
<th>Contracts before weaving aspect</th>
<th>Contracts after weaving aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_{\text{pre}} )</td>
<td>( m_{\text{post}} )</td>
</tr>
<tr>
<td>( m_{\text{post}} \rightarrow \text{aft}_{\text{pre}} )</td>
<td>( \text{aft}<em>{\text{pre}} \rightarrow \text{aft}</em>{\text{post}} )</td>
</tr>
<tr>
<td>( \text{aft}_{\text{pre}} )</td>
<td>( \text{aft}<em>{\text{post}} \rightarrow m</em>{\text{post}} )</td>
</tr>
</tbody>
</table>

After weaving aspect (after advice), detection flow of the above contracts is similar to weaving before advice. The transformed assertion verification program is shown in Fig. 11.

### D. Weaving Before Advice and After Advice

Aspect definition contains both before advice and after advice, which can be regarded as combination of weaving before advice and after advice into the base program simultaneously. The base program and aspect code annotated by annotation are shown in Fig. 12.
In order to guarantee that running results of the program satisfy the requirements, the contracts, needed to be satisfied before and after weaving aspects, are shown in Table III. In Table III, the first column is same to Table I and Table II, and the second column is the combination of them. According to the conditions of behavioral subtyping, after weaving before advice and after advice into the base program simultaneously, pre-conditions of the program become weaker while post-conditions become stronger.

### Table III.

<table>
<thead>
<tr>
<th>Contracts before weaving aspect</th>
<th>Contracts after weaving aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>before advice's pre-condition... before(xa)(...)</td>
<td>after(ixa)(...)</td>
</tr>
<tr>
<td>before advice's post-condition...</td>
<td>after(ixa)(...)</td>
</tr>
<tr>
<td>after advice's pre-condition...</td>
<td>after(ixa)(...)</td>
</tr>
<tr>
<td>after advice's post-condition...</td>
<td>after(ixa)(...)</td>
</tr>
</tbody>
</table>

After weaving aspect (before advice and after advice), detection flow of the above contracts is similar to weaving before advice. The transformed assertion verification program is shown in Fig. 14.

What described above is that a single aspect acts on a certain method (join point) of the base program, for multiple aspects acted on a shared join point, in AspectJ, through introducing the statement of declare precedence, to specify the execution priority of advice acting on a shared join point, according to the priority level, weaving aspect one by one is similar to weaving a single aspect.

### IV. TRANSFORMATION OF THE VERIFICATION PROGRAM

This section describes transformation algorithm of the assertion verification (contract checking) program. Take weaving before advice as an example, a specific transformation algorithm is shown in listing 1.
1. Extract contracts description of the base program (classes) and aspect code (advice);
2. Output the code to check whether the pre-condition $a^w_{mp}$ of the aspect is satisfied;
3. Output the code to check whether the pre-condition $(m_{pre} || a^w_{mp})$ of behavioral subtyping is satisfied;
4. Output the code to check whether the post-condition $a^w_{post}$ of the aspect is satisfied;
5. Output the code to check whether weaving aspect breaks pre-condition of the method $m$, namely, whether the condition $(a^w_{mp} || m_{pre})$ is satisfied;
   5.1 Output the code to check whether the post-condition $a^w_{post}$ of aspect is empty or pre-condition $(m_{pre})$ of the method is empty;
   5.3 Output the processing code when both are empty;
6. Output the code to check whether the pre-condition $m_{pre}$ of the method $m$ is satisfied;
7. Output the code to check whether the post-condition $m_{post}$ of the method $m$ is satisfied.

Listing 1. Contract transformation algorithm after weaving before advice

For weaving multiple aspects, first it needs to confirm the amount of aspect and their execution priority, according to the priority level to weave the corresponding aspect. Detection flow of the contracts and the transformation algorithm are similar to weaving a single aspect. It is important to note that when program checks the contracts of weaving aspect i, firstly it needs to judge the weaving type of aspect i-1. Different weaving types may result in changes of different conditions (contracts) in the base program.

V. AN EXAMPLE

Take the program shown in Fig. 2 as an example, after adding the requirement of checking credits, because the system requires that it should play 10 songs at least before weaving Creditsaspect, so pre-condition of the method play (Song) in Jukebox is: user’s credits are larger than or equal to 10, while the post-condition is: user’s credits are larger than or equal to 0. Aspect Creditsaspect is used to check whether the user has enough credits, therefore, its post-condition is: user’s credits are larger than or equal to 0. Jukebox class and Creditsaspect, both annotated by annotation, are shown in Fig. 15.

```
class Jukebox{
    @pre("credits>=10")
    @post("credits>=0")
    public void play(Song song) {
        Player p = new Player();
        p.play(file);
    }
    ...
}

aspect CreditsAspect {
    @post("credits>=0")
    before(): call (public void Jukebox.play (Song)) {
        if (Credits.instance ().enoughCredits()) {
            Credits.instance ().withdraw ();
            System.out.println("Ok! Begin playing...");
        } else {
            throw new NotEnoughCreditsException (j);
        }
    }
}
```

Figure 15. Annotated Jukebox class and Creditsaspect

According to the contract transformation algorithm (listing 1) after weaving aspect, converting contracts of the base program (Jukebox Class) and aspect code (Creditsaspect), the generated assertion verification program is shown in listing 2. It checks the pre-condition of Creditsaspect in line 3, which is empty, so it returns true directly. Then it checks the pre-condition of behavioral subtyping in line 5, in this example, the pre-condition of Creditsaspect is empty while the play method’s pre-condition is not empty, so it returns true directly. In lines 7-9, check the post-condition of Creditsaspect, if it isn’t satisfied, print the related information and return name of the woven aspect; In lines 12-16, it checks whether weaving Creditsaspect breaks the pre-condition of method play; In lines 18-20, check the pre-condition of method play, if it isn’t satisfied, return name of the caller’s method; Finally, in lines 22-24, check the post-condition of method play, if it isn’t satisfied, return name of the callee.

```
1. Class Contract_before{
2.     // check the pre-condition of Creditsaspect (is empty)
3.     return true;
4.     // check the pre-condition of behavioral subtyping
5.     return true;
6.     // check the post-condition of Creditsaspect
7.     if(Credits<0){
8.         System.out.println("violate post-condition of Creditsaspect!");
9.         return aspName;
10.    }
11.    // check whether weaving Creditsaspect breaks pre-condition of the method play
12.    if(Credits<0 && Credits>=10)
13.        return true;
14.    else{
15.        System.out.println("Weaving Creditsaspect breaks pre-condition of the method play!");
16.        return aspName;
17.    }
18.    if(Credits<10){ // check the pre-condition of the method play
19.        System.out.println("violate pre-condition of the method play!");
20.        return callerName;
21.    }
22.    if(Credits<0){ // check the post-condition of the method play
23.        System.out.println("violate post-condition of the method play!");
24.        return methodName;
25.    }
```

Listing 2. Generated assertion verification program after transformation
VI. CONCLUSIONS AND FUTURE WORK

This paper proposes an approach of automatic detection, based on design by contract and behavioral subtyping, to solve the behavioral conflict in AOP application. This approach has the following characteristics:

- Based on ideas of design by Contract and behavioral subtyping, it achieves automatic detection to the behavioral conflict in aspect composition at runtime, and provides safeguard for security composition of the aspects and it is also benefit for building the trusted aspect-oriented software.

- The approach uses Java annotation to describe the contracts in the base program and aspect code, then converts these contracts to the assertion verification program to check whether the program violates the related contracts, consequently achieving automatic detection to the behavioral conflict at runtime.

Our future work will concentrate on: contract checking after weaving around advice, description and verification of the aspects’ invariants, etc.

ACKNOWLEDGMENT

This research project was supported by the National Natural Science Foundation of China under Grant No. 60873024, Provincial Natural Science Foundation of Hubei under Grant No.2009CDB293.

REFERENCES


Chengwan He, born in 1967. In 1997, he received master's degree in Information Engineering of Hokkaido University in Japan. In 2005, he received doctor's degree in Computer Software and Theory of State key laboratory of Software Engineering, Wuhan University, China. He does postdoctoral research in School of Electronic Information, Wuhan University. His current research interests include theory and application of software engineering, knowledge discovery and data mining, intelligent network and software engineering.

Zheng Li, born in 1984. She is studying for a master's degree and her research area is software engineering.