アスペクトの相互作用を解消するアスペクトの提案

武山 文信　　千葉 滋

東京工業大学大学院
情報理工学研究科 数理・計算科学専攻
www.csg.is.titech.ac.jp/~{f_takeyama, chiba}

要 旨

アスペクトの相互作用を解消するために、関連するアスペクトの振る舞いを再定義しなければならない場合がある。従来のようにアスペクトの優先順位を変更するだけでは不十分である。そこで、本論文では相互作用が発生しているジョインポイントに対し、新たにアスペクトを織り込むことで相互作用を解消する方法を提案する。また、これを実現するために、AspectJ を言語拡張し、振る舞いを再定義する `around` アドバイスと、`pointcutOf` ポイントカット、優先順位付き `proceed` を我々は提案する。`pointcutOf` は指定されたアドバイスのポイントカットを表すポイントカットである。また、優先順位付き `proceed` は、指定の優先順位で既存のアドバイスを必要に応じて再利用するための言語機構である。

1　はじめに

アスペクト指向言語を用いた開発では、アスペクトを組み合わせてアプリケーションを開発することがある。このとき、複数のアスペクトが同一のジョインポイントに織り込まれると、アスペクト間に相互作用が発生し、それぞれのアスペクトが意図した動作をしなくなることがある。このようなときに、プログラマは相互作用による不具合を解消できなければならない。

アスペクトの相互作用による不具合を解消するために、相互作用が発生しているジョインポイントで関連するアスペクトの振る舞い全体を再定義しなければならない場合がある。従来の相互作用の解消方法には優先順位の変更が用いられることが多く、AspectJ[4] では `declare precedence` が用意されている。しかし、相互作用を発生させているアスペクトの振る舞いを、同一のジョインポイントに織り込まれている他のアスペクトに応じて変えなければならないときは、優先順位を変更するだけでは相互作用による不具合を解消することができない。

そこで、我々はアスペクトを用いて相互作用を解消する方法を提案する。この方法では、相互作用が発生しているジョインポイントに新しいアスペクトを最も高い優先度で織り込み、相互作用が発生しているアスペクトを上書きするので、従来の方法では解消できなかった相互作用による不具合も解消できる。

また我々は、この方法を支援するために AspectJ を拡張し、抽象化と既存のコードを再利用するための言語機構を追加する。これによって、従来の AspectJ より抽象的な記述で、アスペクトによる相互作用の不具合を解消を実現できる。

以下、2 章では優先順位による方法では解消できない相互作用の例を示す。3 章では、アスペクトを用いた相互作用の解消方法を提案し、AspectJ に追加した言語機構について説明する。4 章では、2 章で示した以外の相互作用と解消方法の例を示す。5 章では言語デザインに関する議論を行う。6 章では、関連研究について取り上げ、7 章で本論文をまとめる。

2　相互作用の解消

2.1　Shape クラスに作用するアスペクト

相互作用が発生するアスペクトの例を示すため、ここでは簡単な図形エディタを想定し、図形を表す `Shape` クラス（図 1）に織り込まれる 3 つのアスペクトを考える。`Shape` クラスのメソッド `getWidth`、`setWidth` は図形の幅を取得したり、変更したりするメソッドで、`scale` は倍率を指定して図形の幅と高さを変更するメソッドである。

```
public class Shape {
  int width;
  int height;
  public void setWidth(int width) {
    this.width = width;
  }
  public int getWidth() {
    return width;
  }
  public void scale(double scale) {
    width = (int) Math.round(width * scale);
    height = (int) Math.round(height * scale);
  }
  //略
}
```

図 1: Shape クラス

```
public aspect DoubleCoordinate {
  double Shape.dblWidth;
  double Shape.dblHeight;
  void around(double width, Shape shape):
      execution(void Shape.setWidth(int)) &&
      args(width) && target(shape) {
    shape.dblWidth = width;
  }
  int around(Shape shape):
      execution(int Shape.getWidth()) &&
      target(shape) {
    return (int)Math.round(shape.dblWidth);
  }
  void around(double scale, Shape shape):
      execution(void Shape.scale(double)) &&
      args(scale) && target(shape) {
    shape.dblWidth *= scale;
    shape.dblHeight *= scale;
  }
}
```

図 2: DoubleCoordinate アスペクト

まず、図形の高さと幅を浮動小数点数で保持するように拡張する場合を考える。拡張を行うには、クラスのコードを直接変更する方法もあるが、拡張をモジュール化するため、DoubleCoordinate アスペクトでこれを実現する（図 2）。もとの scale メソッドは小数点以下を丸めているが、このアスペクトでは丸めずに保持するので、scale メソッドを繰り返し呼び出したときのフィールド幅と高さの精度が向上する。

次に ReallyChanged アスペクトを考える（図 3）。これは、図形の大きさを変更するために setWidth や scale メソッドが呼ばれたときに、変更の前後でフィールド width などの値が同じである場合は何もしないようにするアスペクトである。後述する ObserverProtocol アスペクトの不要な実行も抑制することで、実行時性能を向上させるためのものである。

最後に ObserverProtocol アスペクトを考える（図 4）。これは、オブサーバーパターンを実装する

```
public aspect ReallyChanged {
  void around(int width, Shape shape):
      execution(void Shape.setWidth(int)) &&
      args(width) && target(shape) {
    if (shape.getWidth() != width) {
      proceed(width, shape);
    }
  }
  void around(double scale):
      execution(void Shape.scale(double)) &&
      args(scale) {
    if (scale != 1.0) {
      proceed(scale);
    }
  }
  //略
}
```

図 3: ReallyChanged アスペクト

```
public aspect ObserverProtocol {
  after(): execution(void Shape.setWidth(int)) ||
          /* 略*/ ||
         execution(void Shape.scale(double)) {
    //observer.notify();
  }
}
```

図 4: ObserverProtocol アスペクト

アスペクトである [3]。エディタ上に描画された図形の大きさが変更されたときに、画面の再描画などを行う。

2.2 アスペクト間の相互作用

これら 3 つのアスペクトが Shape クラスに同時に織り込まれると、それぞれのアスペクトのアドバイスが同一のジョインポイントで実行される可能性がある。このように複数のアスペクトが同一のジョインポイントに織り込まれることでアスペクトの相互作用が発生する。相互作用は、しばしば意図した結果を生まなくなる。相互作用の発生するジョインポイントの例としては Shape クラスの setWidth メソッドの実行時がある。

AspectJ ではそれぞれのアドバイスに対して優先度が与えられる。同じアスペクト内で宣言されたアドバイスは、宣言された位置で優先順位が決定されるが、継承関係のない異なるアスペクトで宣言されたアドバイス間の優先順位は未定義である [1]。仮に DoubleCoordinate、ReallyChanged、ObserverProtocol の順で、それぞれのアスペクトのアドバイスに優先度が与えられるとする。この順序には問題がある。ジョインポイントでアスペクトの相互作用が発生しているとき、まず最も優先度の

```
aspect Precedence {
  declare precedence:
    ReallyChanged, ObserverProtocol, DoubleCoordinate;
}
```

図 5: declare precedence による優先順位の指定

高いアドバイスが選ばれ実行される。アドバイス中で proceed が呼び出されると、次の優先度のアドバイスが実行される。これはクラス継承の super 呼び出しに相当する。最も優先度の低いアドバイスから proceed が呼び出されると、ジョンポイントにあるベースコードの処理が実行される。before アドバイスと after アドバイスは、proceed の前または後に処理を行い、proceed を必ず呼び出すような特殊な around アドバイスと考えられる。例の場合では DoubleCoordinate アスペクトのアドバイスが、setWidth メソッドの実行時に proceed を呼ばないので、他のアドバイスが実行されない問題がある。

2.3 優先順位の変更

どのように優先順位を与えても、相互作用による不具合を解消できない場合がある。AspectJ は declare precedence でアスペクト間の優先順位を明示的に指定する機能を持つ。優先度の高いアスペクトで宣言されたアドバイスは優先度が高くなる。図 5 のように記述すると、ReallyChanged、ObserverProtocol、DoubleCoordinate の順序で実行させることができる。

このとき、DoubleCoordinate アスペクトによってインタータイプ宣言された dblWidth の値が 3.2 であるときに setWidth(3) が呼び出されると、dblWidth が 3.0 になることが期待される。しかし、getWidth メソッドによって得られる値は 3 であり、shape.getWidth() != width が偽であるので、ReallyChanged アスペクトが、setWidth メソッドそのものを無視してしまう。残り 2 つのアスペクトのアドバイスは実行されず、dblWidth の値も 3.2 のままである。

3 アスペクトを用いた相互作用の解消

我々は、相互作用による不具合を解消するために、相互作用が発生しているジョインポイントに新しいアスペクトを織り込むことで、既存のものを上書きする方法を提案する。また、この方法を支援する AspectJ 拡張も提案する。我々の方法では、不具合を生じている箇所を特定し、その箇所についてだけアスペクトを修正することで不具合を解消する。

アドバイス A と B が相互作用により不具合を生じているとき、これを解消するために新しく織り込まれるアドバイス C は次のようなものである。まずアドバイス C が織り込まれるジョンポイントは、A と B が共に織り込まれるジョインポイントである。したがってアドバイス C のポイントカットは、A と B のポイントカットの論理積となる。また C は A と B を上書きするように織り込まれなくてはならないので、C には A と B より高い優先度を与える。C の処理内容は、A と B が実行する処理内容を正しく融合したものである。

このようにすることで、C が織り込まれる以前は A と B が共に織り込まれて不具合が発生していたジョインポイントでは、優先度が最も高い C が代わりに実行され、A と B に相当する処理をおこなう。C が内部で proceed を呼び出さなければ、A と B の処理は決して実行されない。A と B の一方しか織り込まれないジョイポイントには C は織り込まれないので、A あるいは B がそれまで通り実行される。

本方法では、相互作用の解消にもアスペクトを用いるので、既存のアスペクトを編集する必要が無い。アスペクトを編集すると、相互作用が発生してしているジョインポイント以外にも影響を与えてしまう危険性がある。本方法では相互作用が発生しているジョインポイント以外では振る舞いを変更しない。

この方法を支援するために、我々は AspectJ に pointcutOf ポイントカットと優先順位付き proceed 等の言語機構を追加する。また、これらの言語拡張による 2 章の例の不具合を解消する方法を示す。

3.1 アドバイスの識別

相互作用が発生しているジョインポイントの特定のためにアドバイスを識別する方法が必要となる。AspectJ にはアドバイスを識別する方法が無いため、本言語ではアドバイスに名前を付けることでアドバイスを識別する。

アドバイス名はメソッドのようにアドバイスの種類とパラメータの間に記述する。アドバイスの定義はおよそ次の通りである。after アドバイスなどについても同様である。

Type around *Advice-Name*(*Type Variable-Name*,...){ *Body* }

3.2 pointcutOf ポイントカット

相互作用による不具合が発生しているジョインポイントの特定を容易にするため、我々は pointcutOf ポイントカットを提案する。これは、引数で与えられたアドバイスが実行されるジョインポイントを表す。相互作用を解消したいアドバイスが実行されるジョインポイントを pointcutOf でそれぞれ取り出し、それらすべての論理積を取れば、問題のジョインポイントが得られる。アドバイス A と B が相互作用するジョインポイントは pointcutOf(A) && pointcutOf(B) のように記述できる。pointcutOf は if や within など、他のポイントカットと論理積を取ることもできる。これにより、動的な条件で相互作用による不具合が生じる場合にも対応できる。pointcutOf ポイントカットの構文は以下のようになる。

pointcutOf(*Advice-Name*(*TypePattern* or *Id*,...))

pointcutOf ポイントカットの引数のアドバイスはその名前で識別する。そのアドバイスが引数を取るときは、名前に続けてその引数の型（タイプパターン）を書く。型の代わりに引数名を書くこともできる。例えば図 7 では、

```
pointcutOf(DoubleCoordinate.setWidth(w, s))
```

のように、引数名 w と s が指定されている。この場合、w と s の値は、それぞれ DoubleCoordinate アスペクトの setWidth によって決定され、Resolve アスペクトのアドバイス中で利用できる。

3.3 優先順位付き proceed

相互作用による不都合を解消するアスペクトのアドバイスを定義するとき、相互作用を起こしている既存のアドバイスの定義を再利用できるようにするため、我々は優先順位付き proceed を提案する。これにより、プログラマは既存のアドバイスを再度記述する必要が無くなる。構文は以下のようになる。

[*Advice-Name*, ...].proceed(*Expression*, ...)

優先順位付き proceed は、pointcutOf で指定されたされたアドバイスを、指定された優先順位で実行する。poincutOf で指定されたアドバイスのうち、優先順位が指定されなかったアドバイスは実行されない。アドバイス A と B、C があり、アドバイスのポイントカットが pointcutOf(A) && pointcutOf(B) && pointcutOf(C) と定義されているとき、このアドバイスで [A, C].proceed() を呼び出すと、A の次に C の順で高い優先度が与えられ、proceed を呼び出す。アドバイス B は実行されない。

ただし、優先順位付き proceed を記述できるのは around resolving アドバイスだけである。このアドバイスは最も高い優先度が与えられる。また、優先順位付き proceed は、相互作用を起こしている既存のアドバイスに対して優先順位を明示的に指定するだけで不都合を解消できる場合にも有用である。

3.4 相互作用の解消例

まず、2 章で示した不具合を解消するためのアスペクトを示す前に、我々が拡張した AspectJ で記述し直した各アスペクトを図 6 に示す。もとの記述との違いは各アドバイスに名前が付いているだけである。例えば DoubleCoordinate アスペクトは、setWidth アドバイスと getWidth アドバイス、scale アドバイスを持つ。

図 7 は、2 章の不具合を解消するためのアスペクトである。このアスペクトでは、setWidth アドバイスを around resolving アドバイスとして定義している。このアドバイスは、3 つのアスペクトの setWidth アドバイスと onChanged アドバイスの相互作用が発生しているジョインポイントで実行される。このジョインポイントを特定するために使われているのが pointcutOf ポイントカットである。pointcutOf ポイントカットによる抽象化により、

```
public aspect DoubleCoordinate {
  double Shape.dblWidth;
  double Shape.dblHeight;
  void around setWidth(double width, Shape shape):
      execution(void Shape.setWidth(int)) &&
    args(width) && target(shape) {
    shape.dblWidth = width;
  }
  int around getWidth(Shape shape):
      execution(int Shape.getWidth()) &&
    target(shape) {
    return (int)Math.round(shape.dblWidth);
  }
  void around scale(double scale, Shape shape):
      execution(void Shape.scale(double)) &&
      args(scale) && target(shape) {
    shape.dblWidth *= scale;
    shape.dblHeight *= scale;
  }
}

public aspect ReallyChange {
  void around setWidth(int width, Shape shape):
      execution(void Shape.setWidth(int)) &&
      args(width) && target(shape) {
    if (shape.getWidth() != width) {
      proceed(width, shape);
    }
  }
  //略
}

public aspect ObserverProtocol {
  after onChanged():
      execution(void Shape.setWidth(int)) ||
      /* 略*/ ||
      execution(void Shape.scale(double)) {
    //observer.notify();
  }
}
```

図 6: アドバイスに名前を付ける

```
public aspect Resolve {
  void around setWidth(int w, Shape s) resolving:
      pointcutOf(DoubleCoordinate.setWidth(w, s)) &&
      pointcutOf(ReallyChanged.setWidth(int, Shape)) &&
      pointcutOf(ObserverProtocol.onChanged()) {
    if (s.dblWidth != (double)w) {
      [ObserverProocol.onChanged,
       DoubleCoordinate.setWidth].proceed(w, s);
    }
  }
  //略
}
```

図 7: 解消のためのアスペクト

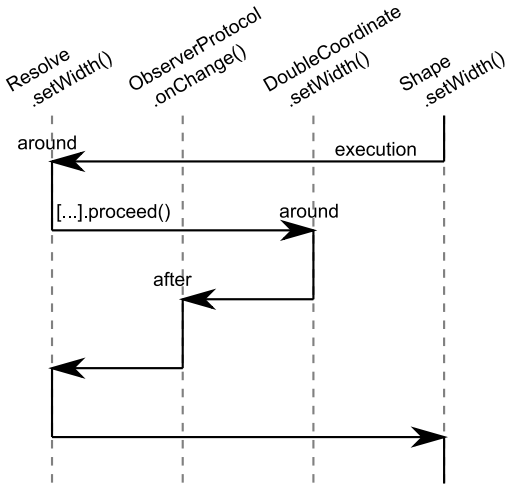

図 8: 実行中のアドバイスの遷移

プログラマは不具合が起きているジョインポイントを、Shape クラスの setWidth メソッドが呼ばれているとき、と具体的に指定する必要が無くなる。

相互作用の解消用のアスペクトは優先順位付き proceed で相互作用を起こしているアドバイスの一部を呼び出す。Resolve アスペクトの setWidth アドバイスは、ObserverProtocol の onChanged アドバイスと DoubleCoordinate の setWidth アドバイスを、この優先順位で実行している。なお ReallyChanged アスペクトの setWidth アドバイスは実行されない。これに相当する処理は、Resolve アスペクトのアドバイス中の if 文で実行されるからである。

我々が提案する拡張を加えた AspectJ においても、優先順位に基づくアドバイスの実行ルールは従来の AspectJ と変わらない。図 6 のアスペクトに、Resolve アスペクトが織り込まれたときの実行順序を図 8 に示す。

4 本言語拡張の使用例

2 章で示したもの以外にもアスペクトが相互作用する場合がある。そのようなアスペクトの例と提案する言語拡張を用いた解消方法の例を示す。

4.1 toString

まず、toString のようなメソッドで、戻り値によって相互作用による不具合が発生する場合がある。例として、アドバイスが織り込まれる MyInteger クラスの toString メソッドは 10 進数の正の整数を半角数字文字の文字列で返すメソッドであるとする。ToStringWithComma アスペクトは数値を 3 桁毎にコンマで区切り文字列として返すアスペクトである（図 9）。ToStringWithFullWidthChars アスペクトは、全角文字しか扱えない処理があるときに、数値を全角で出力するためのアスペクトである（図 10）。

```
public aspect ToStringWithCommas {
  String around toString():
   call(String MyInteger.toString()) || ... {
    String str = proceed();
    StringBuilder builder = new StringBuilder();
    int r = str.length() % 3;
    if (r == 0) {
      r = 3;
    }
    builder.append(str.substring(0, r));
    int i = r;
    while (i < str.length()) {
      builder.append(',');
      builder.append(str.substring(i, i + 3));
      i = i + 3;
    }
    return builder.toString();
  }
}
```

図 9: ToStringWithComma アスペクト

```
public aspect ToStringWithFullWidthChars {
  String around toString():
   call(String MyInteger.toString()) || ... {
    String str = proceed();
    char[] array = str.toCharArray();
    for (int i = 0; i < array.length; i++) {
      switch(array[i]) {
      case '0': array[i] = '０'; break;
      case '1': array[i] = '１'; break;
      //略
      case '9': array[i] = '９'; break;
      }
    }
    return new String(array);
  }
}
```

図 10: ToStringWithFullWidthChars アスペクト

ToStringWithComma アスペクトと ToStringWithFullWidthChars アスペクトは同一のジョインポイントに織り込まれ、相互作用が発生する。ToStringWithCommas アスペクトは半角のコンマを挿入するが、ToStringWithFullChars が織り込まれるときは全角のコンマを挿入しなければならない。

このような相互作用を解消する方法は 2 通り考えられる。どちらの方法でも提案する言語拡張は対応できる。一方は既存のアスペクトを最大限再利用する方法である。この例では、相互作用が発生している既存のアスペクトを呼び出した後に、半角コンマを全角に置き換えることで解消する（図 11）。他方は、既存のアスペクトを再利用せずに、2 つのアスペクトの振る舞いを再定義する方法である。具体的には 3 桁毎に全角カンマを挿入しながら全角数字に置き換える。この方法ではオブジェクトの生成を減らすなどの最適化ができる。

```
public aspect Resolve {
  String around replaceCommas() resolving:
   pointcutOf(ToStringWithCommas.toString()) &&
   pointcutOf(ToStringWithFullWidthChars()) {
    return [ToStringWithCommas.toString,
            ToStringWithFullWidthChars].proceed()
           .replace(',', '、');
  }
}
```

図 11: 半角カンマを全角に置き換えるアスペクト

```
public aspect BasicLogging {
  private Logger logger;
  pointcut logPoint(): ...;
  before log(): logPoint() {
    logger.log(INFORMATION,
               thisJoinPointStaticPart.getSignature() +
              " is called");
  }
}
```

図 12: BasicLogging アスペクト

4.2 Logging

1 つの横断的関心事を 1 つのアスペクトとしてではなく、複数のアスペクトに分けて実装すると、よりモジュール化できる。このようにすると、相互作用による不具合が発生することがあるが、我々が提案する言語拡張により解消できる。次に示す BasicLogging アスペクトは簡単なロギングを実現するアスペクトで、logPoint() ポイントカットで指定したジョインポイントが実行されたときに、そのジョインポイントのシグネチャを出力するものである。

実際に運用すると、より多くのログ情報が必要になることがある。このとき、BasicLogging のコードを書き換えるのが単純であるが、ここでは BasicLogging を書き換えずに、pointcutOf ポイントカットを用いて、新たな情報を出力する AdvancedLogging アスペクトを log アドバイスが実行されるジョインポイントに織り込むとする（図 13）。この例では call ポイントカットと論理積を取り、特定のメソッドが呼ばれたときに追加情報を出力する。

BasicLogging アスペクトと AdvancedLogging アスペクトの間に同期処理を追加するため AddSynchronization アスペクトを定義する（図 14）。BasicLogging アスペクトと AdvancedLogging アスペクトにより、同一のジョインポイントでのログが 2 つの項目として、次のように出力される。

```
public aspect AdvancedLogging {
  private Logger logger;
  before log(String arg):
  pointcutOf(BasicLogging.log()) &&
  call(void java.io.Writer+.write(arg)) {
    logger.log(INFORMATION, "with arguments: " + String)
  }
}
```

図 13: AdvancedLogging アスペクト

```
aspect AddSynchronization {
  before lock():
  pointcutOf(BasicLogging.log) &&
  pointcutOf(AdvancedLogging.log) {
    Logger.lock();
  }
  before unlock():
  pointcutOf(BasicLogging.log) &&
  pointcutOf(AdvancedLogging.log) {
    Logger.unlock();
  }
  around log():
  pointcutOf(BasicLogging.log) &&
  pointcutOf(AdvancedLogging.log) resolving {
    [lock, BasicLog.log, AdvancedLog.log, unlock]
      .proceed();
  }
}
```

図 14: AddSynchronization アスペクト

```
INFORMATION java.io.Writer.write(String) is called.
INFORMATION with arguments:[document1.fig, --debug]
```

1 行目が BasicLogging によって出力された項目で、2 行目が AdvancedLogging によって出力されたものである。しかし、マルチスレッドアプリケーションでは、これら 2 つのログ項目の間に、別のジョインポイントの項目が入る可能性がある。AddSynchronization アスペクトでは、同期を行う lock アドバイスと unlock アドバイスを定義し、優先順位付き proceed でそれぞれのアドバイスを 2 つのロギングアスペクトの前後で実行する。

5 言語デザインに関する議論

我々が提案しているアスペクトを用いた相互作用の解消は従来の AspectJ でも実装可能であるが、記述が煩雑、冗長になる。ここでは、本言語で提案する新たな言語機構の必要性について考える。

```
public aspect Resolve {
  void around setWidth(int w, Shape s) resolving:
    execution(int Shape.getWidth()) &&
    args(width) && target(shape) {
    if (s.dblWidth != (double)w) {
      shape.dblWidth = width;
      //observer.notify();
    }
  }
  //略
}
```

図 15: 従来の AspectJ による記述

5.1 相互作用が発生しているジョインポイントの特定

pointcutOf ポイントカットを用いなくても、相互作用が発生しているジョインポイントを従来の AspectJ が持つポイントカットで直接記述することができる（図 15）。しかし、このように記述すると、pointcutOf による抽象化の恩恵を得られないので、相互作用を発生させているアドバイスのポイントカットの定義が変更された場合に、相互作用を解消するアスペクトの定義を修正しなければならない。

この問題を解決するために、名前付きポイントカットのポイントカット名を用いる方法も考えられる。このためには、名前付きポイントカットの使用を制限する必要があり、我々は言語の単純化のためにアドバイスに名前を付け、pointcutOf ポイントカットで参照する方法を採用した。これは AspectJ の名前付きポイントカットが、他のポイントカットと組み合わせて使われたり、アドバイス間で共有される場合があり、アドバイスと一意に関連づけられないからである。例えば次の例では、名前付きポイントカット pc が 2 つのアドバイスで共有されており、さらに before アドバイスが実行されるジョインポイントは pc がポイントカットするものより多くなっている。

```
pointcut pc(): execution(Shape.setWidth(...));
before(): pc() || execution(Shape.scale(...)) {}
after(): pc() {};
```

5.2 既存のアスペクトの上書きと再利用

優先順位付き proceed が使えないと、相互作用を発生しているアスペクトのうち、一部のアスペクトだけを取り除いて残りのアスペクトのアドバイスを実行したいとき、記述が煩雑になる。優先順位つ

き proceed が使えるなら、[..] の中に実行したいアドバイスだけを書き、実行したくないアドバイスを書かなければよいだけである。しかし使えない場合、相互作用を解消するアスペクトのアドバイスの中では proceed を一切呼ばず、代わりに実行したいアドバイスの処理内容を直接その中に書き下さなければならない（図 15）。これは冗長な記述であり、相互作用を発生しているアスペクトのアドバイスの内容が修正されたときには、解消するアスペクトの定義も一緒に修正しなければならない。別なやり方としては、実行したくないアドバイスのポイントカット定義を修正し、相互作用による不具合が発生しているジョインポイントでは、そもそもそのアドバイスが実行されないようにする手法もある。しかしこの手法は、相互作用を発生している既存のアスペクトの定義を編集しなければならない点が、アスペクトの再利用性の観点から問題である。

6 関連研究

6.1 Context-Aware Composition Rules

Context-Aware Composition Rules[5][6] ではポイントカットに対してルールを定義し、その中でアスペクトではなくアドバイスの優先順位に関する制約を書くことができる。また、アドバイスを実行しないという制約も書ける。さらに、動的なコンテキストによって優先順位が変化する場合には、if ポイントカットを用いて動的にルールを変更することができる。

しかしながら、2 章で示した ReallyChanged アスペクトと DoubleCoordinate アスペクトの例のように優先順位だけでは解消できない相互作用がある。本言語においても、アドバイス単位での優先順位の設定が可能であり、if ポイントカットなどを用いた動的なコンテキストによる優先順位の変更も可能である。

6.2 JAsCo

JAsCo[8] はコンポーネントベース開発向けのアスペクト指向言語である。JAsCo でも我々が提案する解消方法が実現可能であるが、抽象度が低く記述が煩雑になる。JAsCo ではコネクタ内で、アドバイスに相当するフックを具体的なポイントカットを与えて初期化し、フック間の優先順位を設定できる。さらに、実行時にジョインポイントに織り込まれたフックをリストとして取得し、リスト操作することでフックの実行順序を動的に制御できる。

6.3 Traits

AspectJ のような言語における around アドバイスとベースコードの関係は mixin 継承における mixin とクラス間の関係と類似している。around アドバイスで proceed を呼び出すと、次の優先順位のアドバイスまたはベースコードが呼び出されるように、mixin で super を呼び出すと、継承パス上の次の mixin のメソッドまたはクラスメソッドが呼び出される。同じクラスに対して複数の mixin を組み込ませると、相互作用により不具合が生じることがある。

したがって AspectJ におけるアスペクト間の相互作用の問題は、mixin 継承における mixin 間の相互作用の問題に似ており、我々が提案した方法は、traits[7] が trait を導入して mixin 継承の問題を解決した方法に対応する。trait は mixin を改良した新たな構成単位である。複数の trait が同じメソッドを提供し干渉が発生した場合は、クラスのメソッドで相互に干渉しているすべてのメソッドを上書きしなければならない。我々の言語拡張でも干渉が発生したジョインポイントにアスペクトを織り込んで、相互作用が発生しているアドバイスを上書きする。この点で traits と同じである。

6.4 Mixin-Based Aspect Inheritance

Mixin-Based Aspect Inheritance[2] は、アスペクトの継承に mixin を導入することで、後からアスペクトの振る舞いを変更できる。アドバイスをメソッドのように扱うためにアドバイスに名前を付ける点と、既存のアスペクトの振る舞いをアスペクトで上書きする点では我々の提案と同じである。しかし、我々の提案では、複数のアドバイスの振る舞いを同時に上書きできる。

6.5 Doxpects

Doxpects[9] は XML ドキュメントのためのアスペクト指向システムである。アドバイスは XML 要素の置き換えを行う。

Doxpects では、あるアドバイスが、別のアドバイスに依存するとき、そのアドバイスを明示的に宣言できる。これを利用すると、アドバイス間の優先順位を固定することができる。しかし、この方法では、あらかじめ各アドバイスが依存するアドバイスを宣言しなければならないので、新たなアドバイスを追加したときに既存のアドバイスを変更しなければならない。

7 まとめ

本論文において我々は、相互作用が発生しているジョインポイントに新しいアスペクトを織り込むことで、相互作用による不具合を解消する方法を提案した。これによって、従来の優先順位による解消方法では解決できなかった相互作用による不具合を解消できる。また、この方法を支援するために AspectJ を拡張し、`pointcutOf` ポイントカットと優先順位付き `proceed` を追加することを提案した。

本論文で提案した言語拡張はまだ実装されていない。これを実装することは今後の課題の 1 つである。また、`pointcutOf` ポイントカットが名前付きポイントカットに含まれる場合、アドバイスの定義に `pointcutOf` が直接表れず、優先順位付き `proceed` でのアドバイスの指定が壊れやすいので、これを解決することも課題である。また、最も高い優先順位で実行される `around resolving` アスペクト同士の相互作用を解消する方法と、非 `private` なインタータイプ宣言の干渉を解決することも今後の課題である。

参考文献

[1] The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/`.

[2] Sven Apel, Thomas Leich, and Gunter Saake. Mixin-based aspect inheritance. In *Technical Report Number 10*, Germany, 2005. Department of Computer Science, University of Magdeburg.

[3] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[5] Antoine Marot and Roel 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.

[6] Antoine Marot and Roel 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.

[7] Nathanael Schrli, Stphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 – Object-Oriented Programming*, pages 327 – 339, 2003.

[8] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. 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*, pages 21–29, New York, NY, USA, 2003. ACM.

[9] Eric Wohlstadter and Kris De Volder. Doxpects: aspects supporting XML transformation interfaces. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 99–108, New York, NY, USA, 2006. ACM.