AOP 言語への織り込みインターフェイスの導入

A Weaving-Interface for AOP Languages

境 顕宏 † 鵜林 尚靖 † 玉井 哲雄 ††
Akihiro SAKAI Naoyasu UBAYASHI Tetsuo TAMAI

† 九州工業大学大学院情報工学研究科
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
†† 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, University of Tokyo
sakai@minnie.ai.kyutech.ac.jp ubayashi@acm.org tamai@acm.org

アスペクト指向プログラミング (AOP: Aspect-Oriented Programming) は，ログ処理等の横断的関心事を分離するモジュール化機構である．AspectJ は AOP を代表する言語であり，横断的関心事をアスペクトとして記述できる．しかし，横断的関心事は本来相対的なものであり，視点を変えると基本的関心事とみなすことができる．本論文では関心事を全てクラスとして記述する AOP 言語として ccJava を提案し，個々の関心事を織り合わせることによりプログラムを合成する．また，関心事の記述と関心事の合成の記述との間に織り込みインターフェイスと呼ばれる新しいインターフェイスの概念を導入する．このインターフェイスを守ってクラスを実装する限り，クラス開発者は織り込みを意識する必要がなくなる．

1 はじめに

アスペクト指向プログラミング (AOP: Aspect-Oriented Programming) とはログ処理等の横断的関心事を分離するためのモジュール化機構である [5][8][9][15].

AOP を代表する言語として AspectJ[2] があり，横断的関心事はアスペクトと呼ばれるモジュールに記述する．アスペクトは，織り込み (weaving) を行うことによって，既存のクラスと合成される．また，AspectJ はジョインポイント機構 (join point mechanism) を採用しており，プログラムの実行位置を表すジョインポイント (join point)，ジョインポイントの中から特定のジョインポイントを選ぶためのポイントカット (pointcut)，ポイントカットにより選び出された箇所に影響を与えるアドバイス (advice) の 3 つの要素からなる．例えば，ログ処理を行うアスペクトを考える．プログラムでログを取りたい実行位置をポイントカットで記述し，ファイルにログを書き込む処理をアドバイスとして記述することにより，ポイントカットで指定した位置にアドバイスの内容を織り込むことができる．このように AOP を用いると，基本的関心事の中に横断的関心事を織り込む処理を簡単に記述できる．

横断的関心事は本来相対的なものであり，視点を変えると基本的関心事とみなすことができるため，横断的関心事をクラスとして再利用したい場合が出てくる．しかし AspectJ では，横断的関心事をアスペクトとして記述している為，それを基本的関心事として再利用することが難しい場合が存在する．

そこで本研究では，横断的関心事と基本的関心事の区別をせず，関心事を全てクラスとして記述する，クラスベースの AOP 言語 ccJava(Class-based Cross-cutting language for Java) を提案する．ccJava は，個々の関心事を織り合わせることにより合成する．また，関心事を記述する側と関心事の合成を記述する側の間で新しいインターフェイスの概念を導入する．このインターフェイスのことを織り込みインターフェイスと言う．織り込みインターフェイスに従ってプログラムを記述する限り，関心事の実装をする側は織り込みを意識せず，関心事の合成をする側は関心事の実装を知らなくてもよい．つまり，これらを分けて開発することが可能になる．

AOP 言語にはクラスを記述する側はアスペクトによる織り込みを意識する必要がない性質があり，Obliviousness(忘れっぽさ) と呼んでいるが，既存の AOP 言語は，関心事を記述する側と関心事の合成を記述する側の間にインターフェイスが存在しなかった．そのため，任意の実行位置を織り込みの対象として選べるが，ポイントカットの記述を間違えると予期しない箇所に織り込みが行われ，意図通りの振る舞い

をしなくなるという場合もある．これに対し ccJava では，織り込みインターフェイスによって織り込みの対象となる箇所を明確にすることで，Obliviousness な性質を保ちつつ予期しない箇所への織り込みを回避することができる．

本論文ではまず 2 節で既存の AOP 言語の問題点をまとめる．その後 3 節で問題を解決するために ccJava について述べ，4 節で ccJava の記述例を示す．5 節では ccJava の実装を述べ，6 節で ccJava について議論し，7 節で関連研究について述べる．最後に 8 節で本論文をまとめる．

2 背景と目的

この節では，AspectJ によって実装された簡易図形エディタの例を示し，その問題点をまとめる．

2.1 簡易図形エディタ

ここでは，簡易図形エディタを考える．エディタには，点と線を表す Point と Line クラス，画面を表す Display クラスが存在する．また，Point と Line のクラスには，内部に座標値等の属性をもっており，set メソッドを通して，値を設定することができる．一方，Display クラスでは update メソッドが用意されており，このメソッドを呼び出すことで，画面の再描画を行うことができる．

UpdateSignaling アスペクトは，図形の座標が変更された際に図形の再描画を行うためのアスペクトであり，座標値変更の箇所に Display.update() を織り込む．

AspectJ での実装例を以下に示す．

```
class Point{
  int x,y;
  public int set(int x,int y){
    this.x=x; this.y=y;
  }
}

class Line{
  int p1,p2;
  public int set(Point p1,Point p2){
    this.p1=p1; this.p2=p2;
  }
}

class Display{
  public static void update(){
    /* (再描画処理) */
  }
}

aspect UpdateSignaling{
  pointcut change():
    execution(void Point.set(int,int)) ||
    execution(void Line.set(Point,Point)) ||

  after(): change(){
    Display.update();
    /* (1) */
  }
}
```

このコード例において UpdateSignaling アスペクトは，Point クラスと Line クラスの set メソッドの位置をポイントカットで選択し，after アドバイスとして，再描画処理の呼び出しを織り込んでいる．

2.2 問題提起

2.2.1 AspectJ での問題点

AspectJ では基本的関心事はクラス，横断的関心事はアスペクトに記述する．先ほどの例題では図形の属性の設定を基本的関心事，画面の再描画を横断的関心事と捉えていた．しかし，AspectJ には 2 つの問題点がある．

1 つめは，アスペクトの再利用性の問題である．横断的関心事と基本的関心事は相対的なものであり，横断的関心事も見方によっては基本的関心事と見なすことができる．そのため，アスペクトをクラスとして再利用したいというケースも発生するが，AspectJ ではアスペクトをクラスとして再利用したい場合，関心事をそのまま再利用することができない場合がある．図形エディタの例では，アドバイスは画面再描画のメソッド呼び出しだけであるが，(1) で示したアドバイス内に直接処理を記述することも可能である．この場合，基本的関心事としてそのままの形で再利用できず，(1) に記述したコードをクラスとして再利用したい場合は，アドバイス本体をメソッドとして持つクラスを新たに定義する必要がある．よって，横断的関心事をアスペクトとして記述する方法は必ずしも良いとは言えない．

2 つめの問題は，横断的関心事の影響が及ぶ範囲が分かりにくいという問題である．AspectJ は，クラスを記述する側はアスペクトのことを意識せずに，アスペクトを書く側はクラスの実装の詳細を知らなくてもよいという obliviousness と言われる性質をもっている．例として，プロパティーベースのポイントカット表記を用いた場合を考える．図形エディタにおいて，

Point.set(int,int) と Line.set(Point,Point) の部分を*.set(..) と記述すると全てのクラスの set メソッドが織り込み対象になり，値の設定をした時に織り込みを行うことを意味する．しかしこの方法は，メソッド名が一致している関係のないクラスが存在したとき等，織り込みが予期せぬところで織り込みが行われてしまう可能性がある．この例から分かるように，AspectJ で記述したプログラムは，アスペクトが織り込まれる際に実際どこに織り込まれるのかが分かりにづらくなる問題点が生じる場合がある．

2.2.2 クラスベース AOP のアプローチと課題

1 つ目の問題を解決する手法として，関心事を全てクラスで記述するクラスベース AOP がこれまで提案されてきた．ここでは，その例として C#上でクラスベース AOP を実装した EOS-U[7] を取り上げる．

```
public class UpdateSignaling {
 Line l; Point p;

 public UpdateSignaling(Line l, Point p){
   this.ls = ls; this.ps = vs;
 }
 /* アドバイス */
 public void SetShape(){
   Display.update();
   /* (1) */
 }
 /* ポイントカット */
 after execution(
   public void Line.set(Point,Point))
   : call SetShape(); /* (2) */
 after execution(
   public void Point.set(int,int))
   : call SetShape();
}
```

クラスベース AOP 言語では，各関心事をクラスとして記述し，それらを合成することにより横断的関心事の織り込みを実現する．EOS-U の場合，クラスとアスペクトを統合した Classpect と呼ぶモジュールの単位を新たに導入し，関心事を記述する．ここで示すプログラム例は，先ほどの図形エディタを EOS-U のコードで表したもので，UpdateSignaling は Classpect で記述されている．Classpect ではクラスの中にポイントカットを記述できる．プログラムがポイントカットで記述したジョインポイント到達時にアドバイスが実行される．またアドバイスはクラスのメソッドで記述し，実行するアドバイスの指定は (2) のようにポイントカット内で記述する．クラスベース AOP では，AspectJ の時とは異なり，(1) に関心事を実装した場合も，基本的関心事として再利用することが容易である．しかし，第 2 の問題に対しては依然未解決のままである．

これに対して，Hyper/J[13][14] は個々の関心事は通常のクラスとして作成し，クラス同士を合成するルールをメソッドとメソッドのマッチングを記述することにより織り込みを行う．この方法は，合成する箇所を明示的に指定する必要があるため，織り込み先の特定が容易である．しかし Hyper/J は，合成ルールがメソッド名同士のマッチングのみであるため，記述能力という点で不十分である．

2.3 本研究のアプローチ

2.2.1 節で述べた 2 つの問題点を解決するために，本論文では ccJava を提案する．

第 1 の問題に関しては，関心事を Classpect のような拡張を行わずに，純粋な Java クラスで記述するアプローチをとる．関心事を表すクラスにはポイントカットやアドバイスの定義は出現しない．

第 2 の問題に対して ccJava では，織り込みに特化したインターフェイスの概念として，織り込みインターフェイスと呼ばれるものを導入する．織り込みインターフェイスを守る限りクラスは関心事の織り込みを意識することなく実装することができる．また織り込みインターフェイスには，公開した箇所だけに織り込みを限定するメカニズムが存在する．つまり，織り込みインターフェイスによって公開されていない箇所には織り込みの影響が及ばないようにすることで，織り込みがどこに行われるかを明確にすると同時に，予期せぬ織り込みを防止することも可能になる．

予期せぬ織り込みを防止することに関しては，AspectJ でもプログラムの記述方法によっては防ぐことができる．例えば，K.Sullivan らの XPI[12] は，アスペクトとクラスそれぞれをいくつかの設計ルールに従って記述することを提案しており，これに従って記述することで予期しない所へ織り込みを回避した形でプログラムを記述できる．しかし，これらのルールはあくまで自然言語により与えられた方法であった．これに対し ccJava では，言語レベルで織り込み対象の制限を導入する．

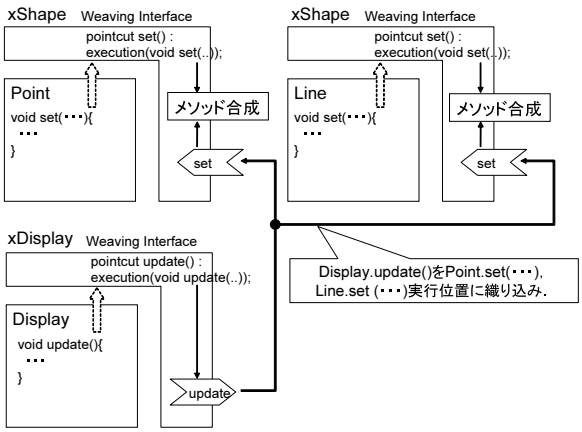

図 1: 織り込みのイメージ

3 ccJava

本節では，クラスベース AOP 言語を Java 言語上で実装した ccJava について説明する．

3.1 概念

ccJava は，関心事を純粋な Java クラスとして記述する．図 1 は，図形エディタの例題に対する，織り込みの様子を表したものである．

プログラムの実行地点等の動的な箇所やクラス内のフィールド名やメソッド名といった静的な箇所を含めた位置をプログラムポイントと呼ぶことにし，ccJava ではプログラムポイント同士のマッチングにより織り込みを行う．この方法は Hyper/J に近いが，Hyper/J はメソッド同士のマッチングであったため，記述能力が低かった．しかし，ccJava はプログラムポイント同士のマッチングであり，AspectJ や EOS-U と同じ記述能力をもっている．

また ccJava では，クラス内にポイントカットは記述せず，織り込みインターフェイスの中で記述する．この織り込みインターフェイスには，ポイントカットとアドバイスが存在し，次のような定義になっている．

ポイントカット 公開するプログラムポイントを指定することであり，織り込み対象となるプログラムポイントを限定する．またプログラムポイントの記述方法は表 1 に示されており，基本的に AspectJ のポイントカット表記に従っている．`class`，`method`，`field` ポイントカットは AspectJ のインタータイプ宣言に似ており，クラスの定義を変更するために使われる．

アドバイス アドバイスは，関心事同士を織り合わせる糊 (glue) のようなものであり，プログラムポイントとプログラムポイントを合成するルールを指定する．アドバイスでは，AspectJ アドバイスの表記が使用できる．例えば，`before` アドバイスの場合は，関心事 **A** が他の関心事 **B** によって影響を受ける際に，**A** が **B** より先に実行されることを意味する．

織り込みは，プログラムポイントとどのプログラムポイントを結合するかを指定するマッチングルールに従って，ポイントカットにより公開されたプログラムポイント同士からジョインポイントを生成し，プログラムの構造や振る舞いを変更する．ccJava で言うジョインポイントとは，織り込み前のプログラムではなく織り込み後のプログラムにおいて，織り込みが行われた位置のことを意味し，3 パートフレームワーク [11] で述べられているジョインポイントの定義に似ている．

また，ポイントカットで公開するプログラムポイントは，インポート (import) とエクスポート (export) の 2 種類が存在する．

エクスポート 他の関心事に影響を与えるために公開するプログラムポイントである．

インポート 他の関心事からの影響を受けるために公開するプログラムポイントである．

この概念を，先ほどの図形エディタを用いて説明する．`Point`，`Line` クラスで使われている `xShape` 織り込みインターフェイスでは，属性を変更した際に画面を再描画させる処理を織り込むので，`set` メソッドの実行位置をポイントカットとして記述する．さらに，インポート宣言を記述する．

一方，`Display` クラスで使われている `xDisplay` 織り込みインターフェイスでは，画面の再描画処理を他の関心事の中に織り込むので，`update` メソッドの実行位置をポイントカットとして記述する．さらに，エクスポート宣言を記述する．

次に，図 1 中の矢印で表してるように，エクスポート宣言されたプログラムポイントと，インポートされたプログラムポイントを繋げることによってジョインポイントを発生させ，織り込みを行う．この例では，画面の再描画処理と図形の属性変更でジョインポイントが生成され，`update` メソッドと `set` メソッドを合成する．

	要素	説明
ポイントカット	AspectJ ポイントカット	AspectJ のポイントカットと似た表記 (execution，call 等)
	class	クラスを選択するポイントカット
	method	メソッドを選択するポイントカット
	field	フィールドを選択するポイントカット
アドバイス	AspectJ アドバイス	AspectJ のアドバイスと同じ (before，after 等)
	introduce	class，method，field ポイントカットと共に使用

表 1: ポイントカットとアドバイス

3.2 織り込みインターフェイス

織り込みインターフェイスは通常の Java のインターフェイスと異なり，ポイントカットやアドバイスといった関心事の合成に関わる情報を構成要素として持つ．表 2 は，Java のインターフェイスと織り込みインターフェイスの関係をまとめている．

Java インターフェイスは，実装をカプセル化する．クラスを利用する側から見ると，内部の実装を知らなくても使え，逆に実装する側はクラスがどう使われるかを意識しなくてもよくなる．これに対して織り込みインターフェイスは，織り込み対象となるプログラムポイントをカプセル化する．関心事の合成をする側は，プログラムポイントがどう実現されているかを知らなくても織り込みができ，逆に関心事を実装する側は織り込みを意識する必要がない．

また，Java インターフェイスを実装するクラスでは，カプセル化対象のメソッドを実装しなければならないのと同様に，織り込みインターフェイスを実装するクラスではポイントカットで示したプログラムポイントが存在している必要がある．

3.3 言語仕様

ccJava の言語仕様は次のようになっている．また，具体的な記述方法は次節で例を挙げて説明する．

```
ccJava_program
  ::= (weaving_interface)* weave_statement

weaving_interface
 ::= (modifier)* "w_interface" identifier
     ("extends" (identifier)+)?
     "{" field_declaration "}"
field_declaration
 ::= (pointcut_declaration |
      import_declaration |
      export_declaration)*
pointcut_declaration
 ::= < same as AspectJ's pointcut,
         class, field and method >
import_declaration
 ::= "import" advice_declaration ";"
advice_declaration
 ::= pointcut_name
     < same as AspectJ's advice
         and introduce >
export_declaration
 ::= "export" pointcut_name identifier ";"
pointcut_name
 ::= < same as AspectJ's pointcut name >

weave_statement
 ::= "weave" "{"
     (implement_statement)*
     (connect_statement)* "}"
implement_statement
 ::= "class" identifier "implements"
     identifier ("replacing" identifier
     "with" identifier)*
     ("factory" identifier)? ";"
connect_statement
 ::= "connect" "("
      import_name "," export_name ")" ";"
import_name
 ::= weaving_interface_name "." pointcut_name
export_name
 ::= weaving_interface_name "." pointcut_name
weaving_interface_name
 ::= identifier
```

ccJava プログラムの主要な要素は文法定義で表すと，weaving_interface と weave_statement の部分である．それぞれ，織り込みインターフェイスと織り込みの定義であり，次のようにまとめることができる．

織り込みインターフェイス定義 織り込みインターフェイスの定義を行う部分である．インターフェイス内には，ポイントカットやアドバイス，インポートとエクスポート対象を記述する．

織り込み定義 織り込み定義は大きく 2 つの要素が含まれる．1 つ目は実装の定義である．ここには，どのクラスがどの織り込みインターフェイスを

	構成要素	クラスが実装すべき内容
Java インターフェイス	メソッド定義	メソッドの中身
織り込み インターフェイス	ポイントカット，アドバイス，インポートとエクスポート対象のポイントカット	ポイントカットで示したプログラムポイント

表 2: Java インターフェイスとの関係

実装するのかを記述する．2 つ目はマッチングルールの定義である．ここでは，織り込みインターフェイスがエクスポートしたものを，他のどの織り込みインターフェイスがインポートするのかを定義する．ccJava コンパイラはこのルールに従って織り込みを行う．

4 記述例

この節では，いくつかの例題を用いて ccJava の記述例を示す．

4.1 図形エディタ

ここでは，図形エディタの例題を ccJava で記述する．関心事を表すクラスは 2 節で述べた Point，Line，Display の 3 つのクラスを用いる．

初めに，図形クラスの織り込みインターフェイスを考える．図形クラスの織り込みインターフェイス xShape は，set メソッド実行時に画面再描画の関心事を織り込ませる必要がある．しがたって，set メソッドの実行位置をポイントカットとして選択し，他の織り込みインターフェイスからの織り込みを可能にする為にインポートをする．

```
w_interface xShape{
  pointcut set() : /* (1) */
    execution(void set(int,int)) ||
    execution(void set(Point,Point));
  import after set(); /* (2) */
}
```

コード中の (1) は，クラス内のプログラムポイントから set メソッド実行位置を切り出すためのポイントカットで，(2) より set ポイントカットで指定したプログラムポイント実行後において，他の関心事をインポートする．ポイントカットの表記は AspectJ と似ているが，若干異なる点がある．図形エディタ例で，AspectJ では Point.set(int,int) であったが，ccJava ではクラス名を書かずに set(int,int) とする．

また，(2) の部分は around アドバイスを用いて次のように記述することもできる．

```
  import around set(){
    this.proceed(); /* (3) */
    import.proceed(); /* (4) */
  }
```

この記述の場合，around アドバイスは AspectJ と同様，織り込み先のプログラムポイントを置換する．(3) により置換された本来のプログラムポイントを実行した後，(4) により他の織り込みインターフェイスより取り込んだプログラムポイントを実行する．

続いて，画面クラスの織り込みインターフェイスを考える．図形クラスは，画面再描画のメソッド実行位置を他のクラスに対して織り込み可能にする必要がある．

```
w_interface xDisplay{
  pointcut update() : /* (1) */
    execution(void update());
  export update(); /* (2) */
}
```

コード中の (1) は，update メソッド実行位置を切り出すためのポイントカットで，(2) は，切り出したメソッドを他のクラスに対して公開するためのエクスポートの宣言である．

最後に，weave 節を記述する．この節では，どの織り込みインターフェイスをどのクラスが実装するのかとマッチングルールを記述する．

```
weave{
  /* (1) */
  class Point implements xShape;
  class Line implements xShape;
  class Display implements xDisplay;

  /* (2) */
  connect(xDisplay.update, xShape.set);
}
```

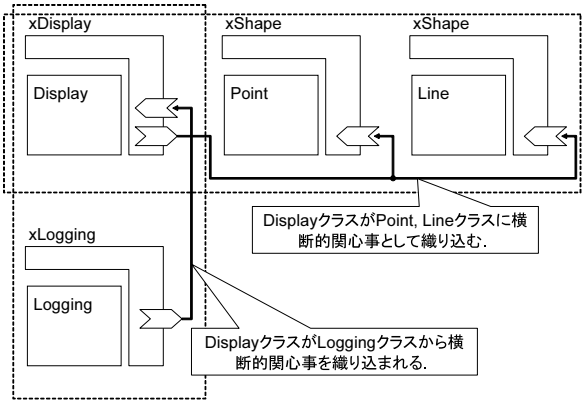

図 2: ログ処理が追加された図形エディタ

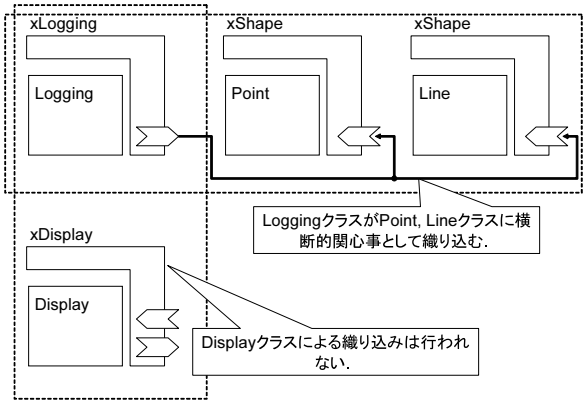

図 3: 属性変更時にログを取るように変更

weave 節中の前半部分 (1) では，どのクラスにどの織り込みインターフェイスを実装させるかのリストを記述している．例えば，Point クラスは xShape 織り込みインターフェイスを実装させている．また，後半部分 (2) は，xDisplay 内の update プログラムポイントを，xShape 内の set プログラムポイントへ織り込むジョインポイントを生成することを意味する．

4.2 ログ処理の追加例

4.1 節の例では，画面クラスの update メソッドは横断的関心事として作用した．ここで，画面の更新をログで記録する処理を追加したい場合，ログ記録の関心事を画面クラスに織り込むことで実現できる．つまり，図 2 のように，画面クラスは画面再描画が横断的関心事として作用し，同時にログ処理の影響を受ける基本的関心事としても作用する．

ccJava では，横断的関心事を意識せずにクラスの実装ができるため，この変更を行う際に，画面クラス内部を変更する必要はない．この処理を追加する手順は，まずログ処理のクラスと織り込みインターフェイスを開発し，xDisplay インターフェイスの update ポイントカットにインポート可能フラグを追加する．さらに，weave 節に，xLogging を実装するクラスとマッチングルールを追加する．

実装例を以下に示す．

```
class Logging {
  public void write(){
    /* ログの書き出し */
  }
}

w_interface xLogging{
  pointcut write() :
    execution(void write());
  export write();
}

w_interface xDisplay{
  pointcut update() :
    execution(void update());
  export update();
  import after update();
}

weave{
  class Point implements xShape;
  class Line implements xShape;
  class Display implements xDisplay;
  class Logging implements xLogging;

  connect(xDisplay.update, xShape.set);
  connect(xLogging.write, xDisplay.update);
}
```

4.3 ログを取る対象の変更

4.2 節の例では，ログのクラスは，画面の更新に対して織り込みを行った．ここで，Point や Line の機能をテストするために，図 3 のように，座標の設定の際に画面の更新をせずにログをとる処理へと変更する．

ccJava では，織り込み先だけを変更する場合は，クラスや織り込みインターフェイスを変更する必要がない．これは，織り込みインターフェイス内で定義するエクスポートやインポートするプログラムポイントに変更がないためである．機能変更をする際は，weave 節内のマッチングルールを図 3 に合うように変更する．実装例を以下に示す．

```
weave{
  class Point implements xShape;
  class Line implements xShape;
  class Display implements xDisplay;
  class Logging implements xLogging;

  connect(xLogging.write, xShape.set);
}
```

AspectJ で織り込み先を変更する際はアスペクト内のポイントカット記述を修正する必要があるが，ccJava を用いると例のように，マッチングルールを書き直すだけで関心事の合成ができる．

4.4 意図しない織り込みの防止

ccJava を用いると，開発者が意図しない織り込みを未然に防止することができる．例えば AspectJ で次のようなコードがあったとする．

```
class Point{
  /* (図形エディタの例と同じ) */
}

class Line{
  /* (図形エディタの例と同じ) */
}

class Display{
  private static boolean lazy=false;

  public static void set(boolean lazy){
    this.lazy=lazy;
    if (!lazy) update();
  }

  piblic static void update(){
    if (!lazy) { /* (1) */
       /* (省略) */
    }
  }
}

aspect UpdateSignaling{
  pointcut change():
    execution(void *.set(..));

  after(): change(){
    Display.update();
  }
}
```

この例は，図形エディタで UpdateSignaling アスペクトが set メソッドの実行位置で，画面の描画処理を呼び出す．この状態で，画面の再描画をコントロールする機能を追加したとする．Display クラスに lazy フィールドを追加し，true の間，画面の再描画を抑制する．さらに lazy 状態は set メソッドを通して変更できるようにし，lazy 状態解除時には，Display クラスの自身の機能により画面の再描画を行う．

このプログラムを実行すると，図形の属性が変化した際に，Display.update() が呼び出されるが，Display.set(false) が実行された時に 2 回画面が再描画される．この問題は UpdateSignaling アスペクトがポイントカットとして，任意のクラスの set メソッドの実行位置を選んでいることに起因する．つまり，Display クラスに set メソッドが追加されたことにより，本来は選ぶべきでないメソッド実行位置までもがポイントカットとして選ばれてしまう．

ccJava では，織り込みが行われる箇所が織り込みインターフェイスで公開されているプログラムポイントのみに限定される．実装例を以下に示す．

```
/* クラス定義は AspectJ の例と同じ．  */

w_interface xShape{
  /* (4.1 節の例と同じ) */
}

w_interface xDisplay{
  /* (4.1 節の例と同じ) */
}

weave{
  class Point implements xShape;
  class Line implements xShape;
  class Display implements xDisplay;

  connect(xDisplay.update, xShape.set);
}
```

ポイントカットの記述は同じ set メソッド実行位置を選んでいるが，xShape は Point と Line のみで実装している．従って，マッチングルールにより画面の再描画が織り込まれる箇所は，織り込みインターフェイス xShape を実装したクラスに限定されるため，lazy 状態解除時に 2 度再描画されることがなくなる．従って，Display.set に対して本来意図しない織り込みが行われる可能性を排除することができる．

5 実装

本研究を進める上で，ccJava コンパイラのプロトタイプを実装した．ccJava コンパイラは AspectJ の

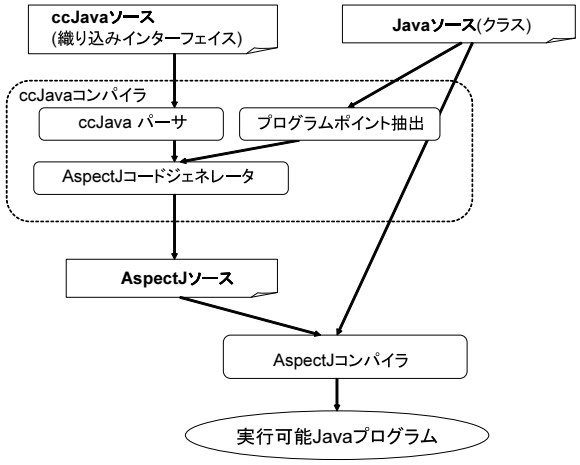

図 4: ccJava 処理手順

へのソースコードトランスレータとして実装されている．

処理の流れを図 4 に示す．まず，ccJava のソースコードを読み取り，AspectJ ソースコードへと変換する．次に，AspectJ コンパイラにより，実行可能な Java プログラムが生成される．

5.1 AspectJ コードの生成

ccJava 処理系が ccJava ソースコードと Java ソースコードを読み取って AspectJ ソースコードを生成する手順を説明する．

1. ccJava のソースコードの `weave` 節を読み取り，織り込みインターフェイス間のマッチングルールを解析する．また織り込みインターフェイスを実装したクラスを特定する．

2. 各 `w_interface` を解析し，ポイントカットとアドバイスを解析する．

3. `w_interface` 内のポイントカットで指定したプログラムポイントが Java コード内に存在するかを調べる．

4. 各 `w_interface` 毎に，AspectJ のアスペクトを生成する．先に調べたマッチングルールを用いて，各アドバイスは `import.proceed()` や `this.proceed()` の呼び出し先を決定する．

5.2 変換プロセス

ここでは，4.1 節で用いた例題プログラムを用いて，実際に AspectJ ソースコードに変換する過程を述べる．

ccJava は，織り込みインターフェイスそれぞれに対してそれと同じ名前のアスペクトを生成する．この例では，`xDisplay` インターフェイスはエクスポートのみを行っており織り込みの対象とはならなため，織り込み対象となる `xPoint` インターフェイスについて述べる．

初めに AspectJ で使用可能なポイントカットを生成する．ポイントカットの表記は基本的に ccJava と AspectJ と同じである為，容易に変換できる．

次にアスペクトから，エクスポートされたメソッドをもつクラスのインスタンスを得る為のコードがアスペクト内に追加される．インスタンスを得る方法は，ファクトリクラスを用いており，ファクトリクラス内の `getInstance` メソッドによって取得されたインスタンスに対して，エクスポートされたメソッドを起動するアスペクトが生成される．ファクトリクラスは，`weave` 節で明示的に指定できるが，省略した場合は次のように自動生成される．

```
/* FileName : DisplayFactory.java */
public class DisplayFactory {
   private static Display instance
      = new Display();

   public static Display getInstance(){
      return instance;
   }
}
```

次にアスペクトのアドバイス本体を生成する．アドバイス本体は，メソッド合成ルールを元に生成される．また，インポートを定義している文法では，アドバイス本体は省略可とされており，省略時のデフォルトの実装は，インポートされたメソッドの起動を行うことになっている．その為，`before` や，`after` アドバイスでは本来のメソッドが実行される前または後にインポートされたメソッドの起動を行う動作が行われる．一方，`around` では本来のプログラムポイントに取って代わりインポートしたプログラムポイントを実行する．本来のプログラムポイントを実行させる場合は，アドバイス本体で `this.proceed()` を用ればよい．

以上の点を踏まえて ccJava により最終的に生成される `xPoint` アスペクトの中身は次のようになる．

```
/* FileName : xShape.aj */
aspect xPoint {
  pointcut onChange() :
```

```
   (execution(void *.set(Point,Point))
    || execution(void *.set(int,int)))
   && (within(Point) || within(Line));

  Display m_display =
    DisplayFactory.getInstance();

  void around() : onChange() {
     proceed();
     m_display.update();
  }
}
```

6 議論

6.1 ccJava と AspectJ の違い

ccJava コンパイラは，第 4 節で述べたように，AspectJ へのソース変換により関心事の織り込みを実現している．したがって，ccJava によって記述できるプログラムは AspectJ によって直接記述することが可能である．一見，ccJava のソースコードを記述し，AspectJ ソースコードへと変換する作業が必要なのだろうかと疑問に思うかもしれない．しかし，第 2 節で述べたように既存の AOP 言語では，関心事の実装時を織り込みを意識せずに行える有効な手段が提供されておらず，さらに予期しない箇所に織り込みが行われる可能性があった．それに対し，ccJava では織り込みインターフェイスにより，プログラムポイントをカプセル化するため，関心事の合成をする側はプログラムポイントの詳細を知らなくてもよい．逆に，関心事を実装する側はどう織り込みが行われるかを意識する必要がなくなる．よって織り込みインターフェイスの概念は，関心事の実装と織り込みを分けて記述する有効な方法の一つであると考えられる．

ccJava は AspectJ を実際に織り込みを行う手段として利用しているが，織り込みインターフェイスを用いて正しく織り込みを行うための検証は ccJava により行われる．最終的に生成されるコードは AspectJ で書けるものと同じであるが設計の手段などは AspectJ とは異なってくる．

6.2 AspectJ における問題点の解決

2.2.1 節では，AspectJ の問題点を 2 つ述べた．第 1 の問題点は再利用性の問題であり，第 2 の問題点は織り込みにおいて影響が及ぶ範囲の問題である．

第 1 の問題点については，クラスベース AOP を用いることで解決した．4.1 節の例題では，画面の再描画は横断的関心事であったが，4.2 節では基本的関心事としてログ処理を織り込ませた．この拡張を実装する際は，記述例で示した通り，クラスに対して手を加えずに修正することができた．

また，第 2 の問題点に関してはマッチングルールを用いることにより解決した．各例題では，マッチングルールによりどのプログラムポイントをどのプログラムポイントに織り込むのかが明確に記述されている．さらに，関心事を記述するクラスは織り込みを意識しない実装になっており，Obliviousness な性質を実現できた．例えば，4.4 節の例題は，`lazy` を解除した場合の再描画は自身の機能で実装されている．一方でこの例題により，不用意な織り込みを防止する織り込みインターフェイスの機能を確認できた．

7 関連研究

この節では，ccJava の概念と関連の深い研究を紹介する．

7.1 クラスベース AOP に関する研究

クラスベース AOP に関する研究として H.Rajan らの EOS-U[7]，IBM の Watson Research Center で研究された Hyper/J[13][14]，S.Chiba らの GluonJ[4][6] がある．

GluonJ は，Java ベースのクラスベース AOP 言語で，アノテーションを用いることで，Java の文法の範囲内で AOP メカニズムを実現している．@Glue アノテーションがついたクラスに，ポイントカットとアドバイス，又はクラスの改良を記述し，コンパイル時又は実行時に織り込みを行っている．

EOS-U と同様にクラスベースである点は ccJava と同じであるが，クラスの中にポイントカット記述が現れてしまう為，第 2 節で述べたものと同じ問題が発生する．

7.2 AOP におけるインターフェイスに関する研究

AOP におけるインターフェイスに関する研究として G.Kiczales らの Aspect-aware interface[7] K.Sullivan らの XPI[12]，J.Aldrich らの Open Module[1] がある．

Aspect-aware interface(AAIF) とは，OOP のインターフェイス内にクラスとアスペクト間の織り込み

の情報を含める手法である．AAIF により，アドバイスがどのメソッドに対して織り込みが行われたのかを表現できる．

XPI は，クラスとアスペクトを設計するためのルール [3] を記述する方法である．その中には情報隠蔽や不用意な織り込み防止に関わるルールが存在する．また，XPI は自然言語により表現されており，そのいくつかは AspectJ のメカニズムにより検証ができる．しかしながら，XPI はルールを方法を提供しているだけである．これに対して，ccJava ではこの機構が言語レベルで利用できる．

Open Module は，アスペクトによる織り込みをクラスによって公開されたプログラムポイントだけに限定し，他のプログラムポイントを隠蔽するメカニズムである．プログラムポイントを隠蔽することにより，不用意な織り込みを防止することができる．ccJava の織り込みインターフェイスは，プログラムポイントの隠蔽を実現している点で，Open Module の考え方を採用している．

これらの研究はは，クラスとアスペクト間のインターフェイスであり，アスペクトを記述する側がクラスの実装を知らなくてもよくなる．これに対して ccJava は関心事中のプログラムポイントをカプセル化し，関心事の合成を記述する側からプログラムポイントの詳細を隠す点で異なっている．

8 まとめ

本論文では，既存の AOP 言語の問題点についてまとめ，その問題点の解決方法として，織り込みインターフェイスの概念を持ったクラスベース AOP 言語 ccJava を提案した．また織り込みインターフェイスにより，プログラムポイントがカプセル化され，関心事の実装と関心事の合成を切り分けることを可能にした．

参考文献

[1] Aldrich, J., “Open Modules: Modular Reasoning about Advice,” *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, to appear, 2005.

[2] AspectJ, http://www.eclipse.org/aspectj/

[3] Baldwin, C., and Clark, K., “Design Rules: The Power of Modularity,” MIT Press, Cambridge, MA, 2000.

[4] Chiba, S., and Ishikawa, R., “Aspect-Oriented Programming Beyond Dependency Injection,” *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pp.121-143, July 2005.

[5] Elrad, T., Filman, R.E., and Bader A., “Aspect-oriented programming,” Communications of the ACM, vol.44, no.10, pp.29-32, 2001.

[6] GluonJ, http://www.csg.is.titech.ac.jp/projects/gluonj/

[7] Kiczales, G., and Mezini, M,, “Aspect-Oriented Programming and Modular Reasoning,” *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp.49-58, 2005.

[8] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J., “Aspect-Oriented Programming,” *Proceeding of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)*, pp.220-242, 1997.

[9] Kiczales, G., and Mezini, M., “Separation of Concerns with Procedures, Annotations, Advice and Pointcuts,” *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pp.195-213, 2005.

[7] Rajan, H., and Sullivan, K., “Classpects: Unifying Aspect- and Object-Oriented Language Design,” *Proceedings of 27th International Conference on Software Engineering (ICSE 2005)*, pp.59-68, May 2005.

[11] Masuhara, H., and Kiczales, G., “Modeling Crosscutting in Aspect-Oriented Mechanisms”, *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, pp.2-28, 2003.

[12] Sullivan, K., Griswold, W., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H., “On the Criteria to be Used in Decomposing Systems into Aspects”, *Proceedings of the 5th joint meetings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, to appear, 2005.

[13] Tarr, P., and Ossher, H., “Multi Dimensional Separation of Concerns using Hyperspaces,” IBM Research Report 21452, April, 1999.

[14] Tarr, P., and Ossher, H., “Hyper/J User and InstallationManual,” IBM Corporation.

[15] 千葉滋, アスペクト指向入門, 技術評論社, 2005.