契約によるクラスとアスペクト間の影響解析

Contract-Based Impact Analysis for Weaving Classes and Aspects

篠塚 卓 † 鵜林 尚靖 † 四野見 秀明 †† 玉井 哲雄 †††
Suguru SHINOTSUKA Naoyasu UBAYASHI Hideaki SHINOMI Tetsuo TAMAI

† 九州工業大学情報工学部
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology
†† 日本 IBM
IBM Japan, Ltd.
††† 東京大学大学院総合文化研究科
Graduate School of Arts and Sciences, University of Tokyo
shinotsuka@acm.org ubayashi@acm.org SHINOMI@jp.ibm.com tamai@acm.org

アスペクト指向プログラミング (AOP) は，ロギングのような横断的関心事をモジュール化する手法である．横断的関心事は織り込みと呼ばれるメカニズムによって既存のクラスと合成され，プログラムの振る舞いが決定される．AOP では，織り込みの仕方に問題があるときプログラムに誤りが混入する可能性が高い．本稿は，AOP 言語の織り込みに対して制約を設ける概念として WbC(Weaving by Contract) を，その記述言語として COW(COntract Writing language) を，各々提案する．COW は，プログラム内の制御フローや依存関係を織り込みに対する制約条件として記述することによって，織り込みによるメソッドの振る舞いの変化を制限できるようにする．COW の制約条件は，プログラムスライシングを用いて検証される．COW は，織り込みがプログラマの意図通り行なわれることを支援するために役立つ．

1 はじめに

アスペクト指向プログラミング (以降 AOP) は，ロギングのように複数のモジュールにまたがって存在する横断的関心事をアスペクトと呼ばれる単位にモジュール化する手法である [8][12][15]．アスペクトは織り込み (weaving) と呼ばれるメカニズムによって既存のクラスと合成され，そのメカニズムはジョインポイントモデル (以降 JPM) によって表現される．JPM はジョインポイント (join point)，ポイントカット (pointcut)，アドバイス (advice) という概念から構成され，プログラム内の様々な位置をジョインポイント，ジョインポイントを選択する記述をポイントカット，ポイントカットが選択した箇所へ織り込まれる影響をアドバイスと呼ぶ．代表的な AOP 言語である AspectJ[2][13] には before，after，around というアドバイスが用意されている．before，after アドバイスは各々ジョインポイントの実行前後に対して織り込みの影響を与え，around アドバイスは織り込み先の処理を置き換える．

AspectJ では織り込みに関する記述はアスペクト内だけに限定でき，織り込み先クラス内には織り込みの記述が現われない．その反面，織り込みを受けているクラスやメソッドの定義だけを見ても，その振る舞いを一概に判断することは出来ない．例えば around アドバイスが織り込まれるとメソッドの処理の一部が置き換わるが，メソッド定義内だけを見ても織り込みの影響は分からない．織り込みの影響を受けているプログラムを修正することは，実装の誤りが混入する可能性を高くする．

本稿では，織り込みがプログラマの意図通り行なわれることを検証する方式として，WbC(Weaving by Contract) を提案する．オブジェクト指向プログラミング (以降 OOP) では，クラスを設計する指針として DbC(Design by Contract)[10][11][18] が広く用いられている．DbC ではメソッド実行の前後に注目して，各々の時点で満たされなければならない条件を事前・事後・不変条件として記述し，これを契約と呼んでいる．これに対し WbC では，織り込みの前後で成立する条件を契約として記述する．

本論文では織り込みの誤りをコンパイル時に発見するため，プログラムの静的側面に着目した WbC を扱う．その手段として，プログラミング言語 COW(COntract Writing language) を提案する．COW ではプログラムスライシング [23][24] を用い

ることにより，制御フローやデータ・制御の依存関係が織り込みでどのように変化するかを表現し，織り込みによるメソッドの振る舞いの変化を検出する．COW を導入することによって，織り込みを受けているプログラムを修正した際の誤り混入を軽減出来る．

以降，本論文では，2 節で AspectJ の織り込みによる問題点を述べる．続いて 3 節では，COW による契約例を示し，COW が織り込みによる誤り混入防止に役立つことを示す．4 節では，COW 言語の処理系構成法について簡単に触れる．5 節で COW について議論し，6 節では本稿に関連する研究と COW の関わりを論じる．最後に 7 節で，本稿をまとめる．

2 問題意識

この節では，AspectJ における織り込みの問題点を G.Kiczales らの例題プログラム [14] を通して具体的に示す．

2.1 AspectJ による簡易図形エディタ

図 1 は，AspectJ で記述された簡単な図形エディタの一部である．Shape を実装した Point や Line はエディタに表示される点や直線である．各々の図形はフィールドとして画面上の描画位置を保持しており，描画位置には各々アクセサが提供されている．moveBy は，アクセサを使わず簡便に図形を移動させるメソッドである．図形の位置が変更されると，UpdateSignaling アスペクトによって Display の再描画が起こる．Display クラスの実装は省略されているが，画面を再描画する update メソッドが定義されている．

図 1 の図形エディタでは，図形を移動させるメソッドが実行されると，実行後に画面の再描画を行なう update メソッドが 1 回呼び出される．プログラマも，織り込み後にそのような update の呼び出しが起こることを意図している．

```
 1: interface Shape {
 2:   public moveBy(int dx, int dy);
 3: }
 4: class Point implements Shape {
 5:   int x, y;
 6:   public int getX() { return x; }
 7:   public int getY() { return y; }
 8:   public void setX(int x) { this.x = x; }
 9:   public void setY(int y) { this.y = y; }
10:   public void moveBy(int dx, int dy) {
11:     x += dx; y += dy;
12:   }
13: }
14: class Line implements Shape {
15:   private Point p1, p2;
16:   public Point getP1() { return p1; }
17:   public Point getP2() { return p2; }
18:   public void moveBy(int dx, int dy) {
19:     p1.x += dx; p1.y += dy;
20:     p2.x += dx; p2.y += dy;
21:   }
22: }
23: aspect UpdateSignaling {
24:   pointcut figureMoved():
25:     execution(void Point.setX(int)) ||
26:     execution(void Point.setY(int)) ||
27:     execution(void Shape+.moveBy(int, int));
28:   after(): figureMoved() {
29:     Display.update();
30:   }
31: }
```

図 1: 簡易図形エディタのプログラム

2.2 修正作業に伴う織り込みの問題点

図 1 に示したプログラムで，Point クラスの x，y フィールドは外部に公開されている．リファクタリング [17] の観点から，これらの可視性は private へと修正されるべきである．しかし可視性の変更は，Point クラス以外の場所で x，y を直接使用しているメソッドに影響を及ぼす．Line.moveBy メソッドではこれらのフィールドを直接使用しており，可視性を修正するためには moveBy の実装は変更が必要となる．Point クラスには x，y のためのアクセサが既に定義されているので，それを用いれば moveBy メソッドは次のように書き直せる．

```
public void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.setY(p1.getY() + dy);
    p2.setX(p2.getX() + dx);
    p2.setY(p2.getY() + dy);
}
```

修正後の moveBy では，p1 や p2 の移動先を設定するために，Point クラスの Setter を使っている．UpdateSignaling アスペクトに注目すると，Point クラスの Setter は Display.update を呼び出すことが分かる．すなわち織り込み後，moveBy メソッドは Point クラスの Setter を使用する度

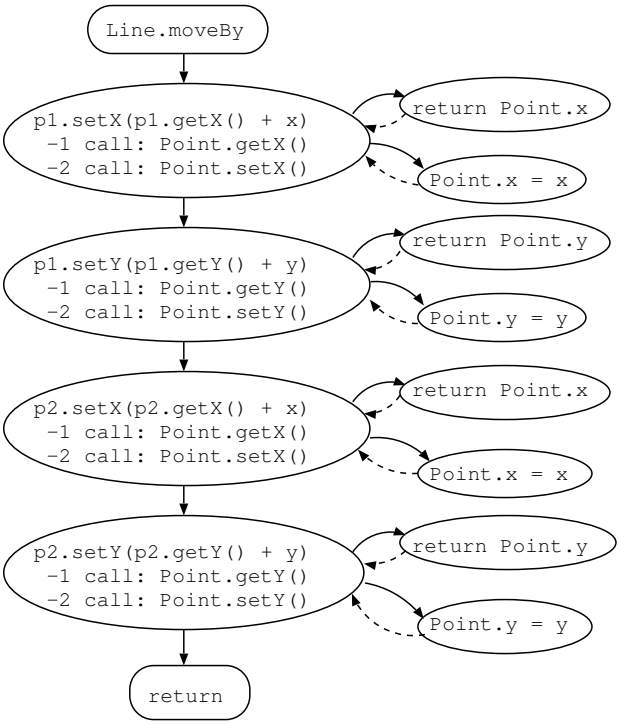


図 2: moveBy メソッドの織り込み前の制御フロー

に Display.update を呼び出すようになる．また，moveBy が実行された後にも Display.update が実行される．従って，修正後のプログラムで moveBy を呼び出すと，Display.update が 5 回も実行されてしまう．

プログラマは，図形が移動される度に 1 回だけ Display.update が行なわれることを意図している．修正後に moveBy 内で Display.update が何度も呼び出されるのは誤りである．この誤りは，AspectJ の織り込みによって生じる．修正の誤りがコンパイル時に発見できなければ，プログラマは誤りをテスト段階まで気付かないかもしれない．

2.3 解決すべき課題

前項のように，織り込みによる影響を受けているプログラムを修正することは，誤りを混入させ易い．このような誤りは，織り込みによってメソッドの制御フローやデータ・制御依存関係が変化することで生じている．例えば，修正後の moveBy メソッドの制御フローは，織り込み前と織り込み後で図 2 と図 3 のように異なる．図 2 と図 3 で異っている点は，太線で示すようなアドバイスを実行する制御フローがあるか否かである．なお，図内の実線は制御フローであり，破

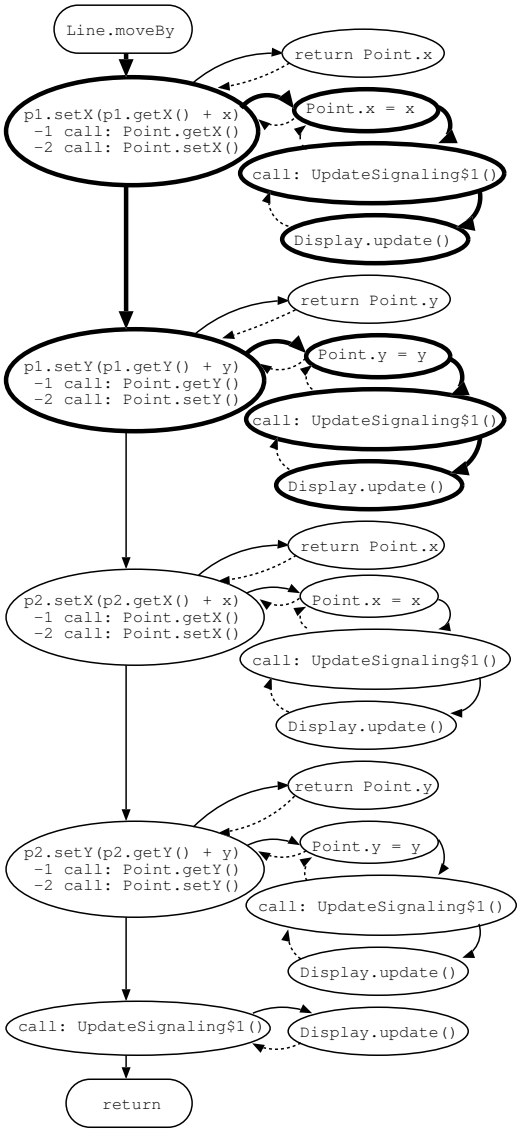


図 3: moveBy メソッドの織り込み後の制御フロー

線はメソッドやアドバイスからの復帰である．円角枠は moveBy メソッドの開始と終了であり，楕円枠はステートメントの実行を意味する．UpdateSignaling$1 はアドバイスである．

このような織り込みの影響をコンパイル時に判断するためには，

1. 制御フロー
2. データの依存関係
3. 制御の依存関係

等を解析する必要がある．これはプログラムスライシングにより可能である．

本稿が提案する COW は，1〜3 を織り込みに対する契約として記述できるようにする．次節以降では，これらを COW で記述する方法について述べる．

3 COW

COW は，制御フローと依存関係によって織り込み時の契約を記述する言語である．ここでは，前節の問題意識を解決する契約を示し，その特性や効果について説明する．

3.1 概念

COW の契約は， 織り込みによって生じるメソッドの振る舞いの変化を予め制約するものである．図形エディタの例では，図形が移動されたときに 1 回だけ画面の再描画が行なわれることが制約である．制約を満たさない修正は誤りである．

図 4 は，図形を移動させる各々のメソッドが内部で 1 回だけ `Display.update` を行なうことを要求する契約である．例えば，28 行目では `Point` クラスの `moveBy` メソッドに対して `Updating.once` という制約を課している．`Updating.once` は，メソッド内で高々 1 回だけ `Display.update` を呼び出すことを表現する．前節の修正では，`moveBy` 内で `Display.update` が何度も実行されるため，契約は満たされない．COW 言語処理系では，契約違反はコンパイルエラーとして扱われる．従って COW によって予めメソッドの振る舞いを規定しておけば，リファクタリング等プログラム修正時に，織り込みによって誤りが生じるのを防ぐのに役に立つ．

3.1.1 事前・事後・不変条件

COW における契約の考え方は，DbC に基づいている．DbC は OOP における設計手法の 1 つであり，プログラムを契約によって設計するという考え方である．DbC は，次の 3 つの概念から構成される．

事前条件 メソッドが呼び出されるための必要条件

事後条件 メソッド実行後に満たされるべき条件

不変条件 メソッド実行前後に関係なく満たされるべき条件

DbC における契約とは，これらの条件を整理することによってプログラムが正しく実行されることを目指すものである．

```
 1: contract Updating {
 2:   define updating(t) {
 3:     statement(t, s) &&
 4:     call(s, void Display.update())
 5:   }
 6:   define plurally(t) {
 7:     statement(t, s1) && statement(t, s2) &&
 8:     controlFlow(s2, s1) &&
 9:     controlFlow(s1s, s1) &&
10:     controlFlow(s2s, s2) &&
11:     !equal(s1s, s2s) &&
12:     call(s1s, void Display.update()) &&
13:     call(s2s, void Display.update())
14:   }
15:   define once(t) {
16:     Updating.updating(t) &&
17:     !Updating.plurally(t)
18:   }
19: }
20: contract ForPoint between
21:   class Point, aspect UpdateSignaling {
22:   void setX(int x) {
23:     ensures(target(t) && Updating.once(t));
24:   }
25:   void setY(int y) {
26:     ensures(target(t) && Updating.once(t));
27:   }
28:   void moveBy(int dx, int dy) {
29:     ensures(target(t) && Updating.once(t));
30:   }
31: }
32: contract ForLine between
33:   class Line, aspect UpdateSignaling {
34:   void moveBy(int dx, int dy) {
35:     ensures(target(t) && Updating.once(t));
36:   }
37: }
```

図 4: 簡易図形エディタ用の COW による契約

COW における契約は，DbC の契約概念を織り込みに対して適用したものである．COW にも事前・事後・不変条件の概念があるが，それらは DbC と以下のような違いを持つ．

事前条件 織り込み前のメソッドの振る舞い

事後条件 織り込み後のメソッドの振る舞い

不変条件 織り込み前後に関係ないメソッドの振る舞い

COW では，織り込みによって生じるメソッドの振る舞いへの作用をこれらの条件で表現する．織り込みの影響を記述することによって，織り込みがプログラマの意図通り行なわれることを支援する．

COW の契約は，織り込みを受けるクラスと，織り込まれるアスペクトとの間に結ばれる．COW では，メソッドの振る舞いに関するプログラム内の静的な性質を，事前事後等の条件として述語論理によって表現する．COW の記述が静的な範囲に限られているのは検証が静的プログラムスライシングに基づいているからである．

述語論理によって表現されるのは，メソッド内の制御フローやステートメントの依存関係である．これらは，メソッドの振る舞いの静的な側面を表現している．COW には，これらを表現するための基本述語が予め用意されており，論理演算で基本述語を組み合わせメソッドの振る舞いを表現する．

3.1.2 契約の記述

COW では，契約を Java のクラス定義に似たスタイルで記述する．例えば図 4 の 20 行目は，

```
contract ForPoint between
    class Point, aspect UpdateSignaling {
```

で始まっているが，この記述は Point クラスと UpdateSignaling アスペクトの間に ForPoint という名前の契約を結ぶことを意味している．

契約で制約するメソッドの振る舞いは，契約定義の {} 内に記述される．COW では，振る舞いを制約するメソッドを指定するために，メソッドのシグネチャを記述する．例えば ForPoint 契約内の，

```
    void setX(int x) { ...
```

という記述は，契約対象である Point クラスの setX メソッドの振る舞いを制約することを表現する．

メソッドの振る舞いに対する制約の内容は，メソッドシグネチャ以降の {} 内に記述される．例えば ForPoint 契約内の setX メソッドに対する制約内容は，

```
ensures(target(t) && Updating.once(t));
```

となっている．ensures は，括弧内の述語論理が事後条件であることを意味する．事前条件を記述するには requires，不変条件を記述するには invariant を使用する．

ensures 内の述語論理で使われている論理変数には，全称や存在が明示されていない．しかし COW で記述出来る論理変数は，存在だけである．述語論理に全称を含めていないのは，論理型言語 Prolog の考え方を COW プログラミングの際にそのまま使えるようにするためである．制約条件

$$target(t)\ \&\&\ Updating.once(t)$$

は，次の論理式

$$\exists t\ [target(t) \wedge Updating.once(t)]$$

と同じ意味を持つ．全称は存在の否定によって表現する．

3.1.3 述語論理によるメソッドの振る舞い表現

上記の制約条件は，制約を受けるメソッドが 1 回だけ Display.update を呼び出すことを指定する．target(t) は，ある変数 t が制約を課されるメソッドであることを意味する．target は，COW に始めから用意されている基本述語である．例えば，ForPoint 契約内の setX メソッドに対する制約条件の記述は，

```
void setX(int x) {
    ensures(target(t) &&
        Updating.once(t));
}
```

のようになっているが，このとき変数 t は，Point.setX メソッドとだけマッチする．

Updating.once は，変数 t のメソッドが 1 回だけ Display.update を呼び出すことを意味する述語である．契約内の述語は，Java のようにドット演算子によってアクセスできる．この述語は COW の基本述語ではなく，Updating 契約の中で新たに定義されている．Updating 契約内の，

```
define once(t) {
    Updating.updating(t) &&
        ! Updating.plurally(t)
}
```

という記述が，既存の述語を組み合わせて新たな述語 Updating.once を定義する記述である．この記述は，

$$Updating.once(t) \equiv Updating.updating(t) \wedge \sim Updating.plurally(t)$$

であることを意味している．

Updating.updating や Updating.plurally 述語も COW の基本述語ではなく，新たに定義されている．Updating.updating 述語の定義は，

```
define updating(t) {
    statement(t, s) &&
        call(s, void Display.update())
}
```

のようになっている．これは，変数 t のメソッドを起点として t から到達できるステートメントが変数 s であり，変数 s のステートメントが Display.update を呼び出しすることを記述している．statement や call は，COW の基本述語である．すなわち Updating.updating は，変数 t のメソッドが内部で Display.update を呼び出しているとき真となる．

Updating.plurally は，次のように定義されている．

```
define plurally(t) {
    statement(t, s1) &&
    statement(t, s2) &&
    controlFlow(s2, s1) &&
    controlFlow(s1s, s1) &&
    controlFlow(s2s, s2) &&
    !equal(s1s, s2s) &&
    call(s1s, void Display.update()) &&
    call(s2s, void Display.update())
}
```

controlFlow は，ステートメント間の制御フローを表わす述語である．この定義は，図 5 のような制御フローを表現する．ステートメントの枠内に記述されているラベルは，Updating.plurally 定義内の変数に対応する．Updating.plurally 述語は，メソッド t が内部で 2 回以上 Display.update を呼び出しているとき真となる．Updating.plurally を定義するために用いられている述語は，全て COW の基本述語である．

表 1 に，COW に用意されている基本述語を示す．なおメソッド，アドバイス，コンストラクタは総じてアクチュエータ (actuator) と称した．プログラマはこれらの述語を組み合わせて新しい述語を定義し，メソッドの振る舞いを表現する．基本述語の組み合わせだけを用いた契約記述は簡単ではないが，COW では Updating のように述語定義を部品化できる．一度記述した述語をライブラリとして再利用することで，契約記述は簡単になる．

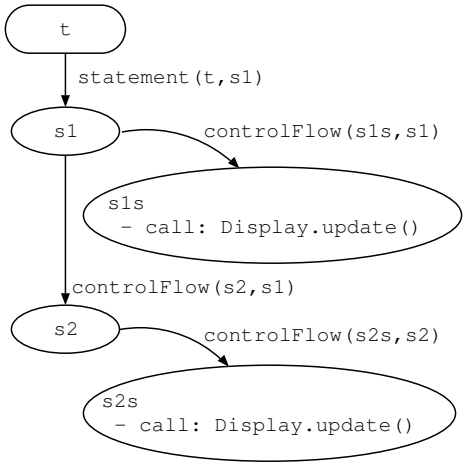


図 5: Updating.plurally が表現する制御フロー

3.2 制御フロー

COW では，メソッドの振る舞いを表現するために制御フローを扱うことが出来る．制御フローとは，例えば図 3 のようなものである．図 3 は，織り込み後の Line.moveBy メソッドの制御フローを表わしている．各々の矢印が制御フローである．COW では，実線矢印を辿って到達可能なステートメント間には全て，制御フローがあると定義する．

Updating.plurally 述語は図 5 の制御フローを表現している．このフローは，例えば図 3 の太線で示した部分グラフとマッチする．

3.3 制御とデータの依存関係

COW では，メソッドの振る舞いを表現するためにステートメント間の依存関係を用いることもできる．次の条件の何れかが満たされるとき，COW ではステートメント間に依存関係があると定義する．

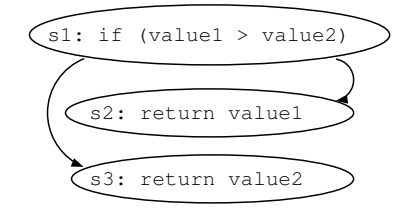
- ステートメント間に制御依存関係がある
- ステートメント間にデータ依存関係がある

3.3.1 制御依存関係

制御依存関係とは，あるステートメントが実行されるか否かが制御文によって変化する場合のことを言う．例えば図 6 に示す最大値を求めるメソッドにおいて，矢印は制御の依存関係を表わす．COW では，矢印を 1 回以上辿って到達できるステートメント間には，全て制御依存関係があると定義する．

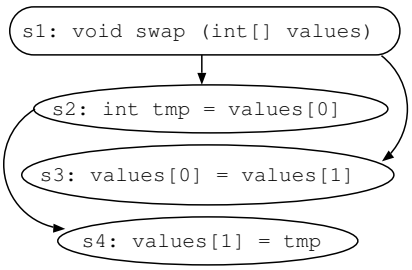
表 1: COW で利用できる基本述語

述語	意味
target(a)	a は織り込み先のアクチュエータ
source(a)	a は織り込み元のアクチュエータ
method(a)	アクチュエータ a はメソッド
constructor(a)	アクチュエータ a はコンストラクタ
advice(a)	アクチュエータ a はアドバイス
statement(a, s)	s はアクチュエータ a を起点に制御フロー上から到達可能なステートメント
entry(a, s)	s はアクチュエータ a で最初に実行されるステートメント
call(s, a)	s はアクチュエータ a を呼び出すステートメント
controlFlow(g, s)	ステートメント s からステートメント g への制御フローがある
controlDependence(d, s)	ステートメント d はステートメント s から制御依存関係を受ける
dataDependence(d, s)	ステートメント d はステートメント s からデータ依存関係を受ける
write(s, f)	ステートメント s はクラスフィールド f に書き込みする
read(s, f)	ステートメント s はクラスフィールド f を読み込む
equal(x, y)	x と y が等しい
true	常に真
false	常に偽




```
int max (int value1, int value2) {
    if (value1 > value2)
        return value1;
    else
        return value2;
}
```

図 6: ステートメント間の制御依存関係




```
void swap (int[] values) {
    int tmp = values[0];
    values[0] = values[1];
    values[1] = tmp;
}
```

図 7: ステートメント間のデータ依存関係

表 1 に示す `controlDependence(d,s)` 述語は，制御依存関係を記述するためのものである．例えば，図 6 で s が s1 で d が s2 か s3 のとき，この述語は真になる．

3.3.2 データ依存関係

データ依存関係は，あるステートメントで参照する変数値が他のステートメントによって定義されていることを言う．例えば図 7 に示す配列のスワップメソッドにおいて，矢印はデータ依存関係を表わしている．1 回以上矢印を辿って到達できるステートメント間には，データの依存関係があると定義する．

表 1 の `dataDependence(d,s)` 述語は，データの依存関係を表現する．例えば，図 7 で s が s1 のときに d が s2 か s3 であれば，この述語は真になる．s が s2 であれば，d が s4 のときだけ真となる．

4 実装

この節では，COW 言語処理系の実現方法について述べる．

```
 1:  controlFlow(Goal, Start) :-                  % Start から Goal への経路を探索する.
 2:      statementTraverse(Start, Goal, [Start]). %
 3:  statementTraverse(Org, Goal, Visit) :-       % Org がメソッドを呼び出すか否かに関
 4:      nextStatement(Org, Next),                % 係なく，Org から Goal への経路を再帰
 5:      \+member(Next, Visit),                   % 的にトラバースする．\+は Prolog の
 6:      append(Visit, [Next], NVisit),           % 否定論理である.
 7:      statementTraverse(Next, Goal, NVisit).   %
 8:  statementTraverse(Org, Goal, Visit) :-       % Org がメソッド呼び出しを行なうステ
 9:      call(Org, Calliee),                      % ートメントで，呼び出し先がまだトラ
10:      entry(Calliee, Entry),                   % バースされていない場合のルール.
11:      \+member(Entry, Visit),                  % call は COW の基本述語である.
12:      append(Visit, [Entry], NVisit),          %
13:      statementTraverse(Entry, Goal, NVisit). %
14:  statementTraverse(Org, Goal, Visit) :-       % Org がメソッド呼び出しを行なうステ
15:      call(Org, Calliee),                      % ートメントで，呼び出し先を既にトラ
16:      entry(Calliee, Entry),                   % バース済みの場合のルール.
17:      member(Entry, Visit),                    %
18:      nextStatement(Org, Next),                %
19:      \+member(Next, Visit),                   %
20:      append(Visit, [Next], NVisit),           %
21:      statementTraverse(Next, Goal, NVisit).   %
22:  statementTraverse(Goal, Goal, _).            % 再帰呼び出しの停止条件.
```

図 8: controlFlow 述語の機能を実現する Prolog ルール

4.1 概要

COW 言語処理系は，大きく分けて次のコンポーネントから構成される．

- COW 構文解析器
- Java/AspectJ ソースプログラム解析器
  - 織り込み箇所解析器
  - プログラム依存グラフ生成器
- 契約検証器

COW 処理系では，先ず COW 構文解析器によって契約記述が解析される．契約検証のために必要な織り込み箇所の特定やプログラムスライシングは，Java/AspectJ ソースプログラム解析器によって行なわれる．契約条件は契約検証器が検査するが，これは Prolog 言語のインタプリタによって実現されている．

4.2 アルゴリズム

COW 言語処理系では，契約記述に用いられた述語論理の充足可能性を判定するために，内部的に Prolog 言語のインタプリタを使用している．ここでは，処理系の中核を担う検証器を実現するアルゴリズムについて説明する．

COW の基本述語の中には，制御フローグラフや制御依存グラフ・データ依存グラフ等をトラバースする述語が含まれている．例えば controlFlow 述語は，controlFlow(Goal, Start) のような使われ方をすると制御フローグラフをトラバースし，Start から Goal へと至る経路があるか否かを判定する．このようなトラバースも，COW では Prolog インタプリタによって行なわれている．controlFlow 述語の機能を実現するためは，図 8 のルールが自動生成される．この Prolog ルールの要点は次の通りである．

controlFlow(Goal,Start) 制御フローグラフをトラバースして，`Start` から `Goal` へと至る経路を探索する．

statementTraverse(Org,Goal,Visit) `Org` から `Goal` へと至る経路を深さ優先で再帰的に探索する．`Visit` には `Org` から `Goal` までの経路情報が書き込まれる．

図 8 で使われている `nextStatement` 述語は，図 9 に示すような制御フローグラフのアークを意味する．フローが述語で表現されることで，再帰的なトラバースが可能となる．トラバースによって経路を発見できれば，controlFlow 述語は真となる．

COW 言語で利用される述語は，このように Prolog のルールや定義へと変換される．変換されたルールは，反駁導出と呼ばれるメカニズムによって，Prolog 内で自動的に検証される．

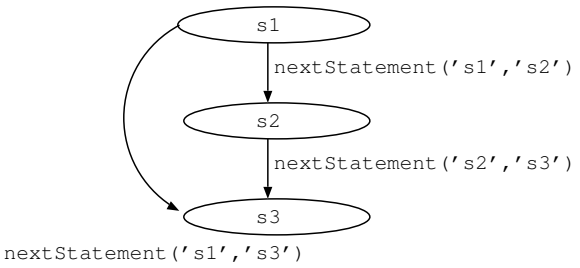


図 9: nextStatement による制御フローの表現

5 議論

ここでは本稿が提案する COW について，いくつかの側面から議論する．

5.1 契約の記述量

現状の COW による契約記述は，プログラマに必要以上の手間を強いるかもしれない．

COW の契約は，クラスとアスペクトの間に一対一関係で結ばれる．従って N 個のクラスと M 個のアスペクトがあるとき，織り込みの影響全てを制約するには少なくとも $N \times M$ 個の契約記述が必要であり，N や M が大きければ記述に手間がかかる．

N+M 個程度の記述が実際にプログラマが実装するモジュール数であるから，契約記述をこの個数程度に抑えられれば記述の手間を少なくできる．これは，1 つのクラスと複数のアスペクトが契約を結べるようにすれば実現できる．クラスの側を 1 つに限定しているのは，COW の契約がメソッドに対して記述されるからである．

一対多関係での契約記述は，例えば次のように，正規表現による型名のマッチングを導入すれば書けるようになる．

```
contract AdvancedContract
    between class Point, aspect *.* {
    ...
}
```

5.2 アスペクト同士の契約

COW における契約は，織り込みで影響を受けるクラスと，そのクラスに影響を与えるアスペクトとの間に結ばれる．この方法だけでは，アスペクトがアスペクトに影響を与える場合に織り込みを制約出来ない．

しかしアスペクト同士が契約を結ぶためには，単純に COW の文法を拡張するだけで済む．例えば，契約記述に使用する between の文法を拡張すれば，アスペクト間に契約を結ぶこと自体はできる．すなわち，

```
contract ForAspects
    between aspect Name1, aspect Name2 {
    ...
}
```

のような記述を可能にすれば良い．

もう 1 つ必要な文法拡張は，アドバイスの定義位置を特定するためのシグネチャ記述を導入することである．例えばメソッドにはシグネチャがあり，メソッドシグネチャを見ればメソッドの定義位置がわかる．しかしアドバイスはシグネチャを持たない．アドバイスに対して契約条件を記述するためには，メソッドシグネチャに相当する表記方法が必要である．必ずしも適切とは言えないが，簡単な方法としては，例えば UpdateSignaling$1 のように，アドバイスがアスペクトの上から何番目に定義されているかを用いることも出来る．

5.3 静的解析の限界

COW の影響解析は，プログラムスライシングに基づいている．そのため，プログラム実行時にしか判断できないようなメソッドの振る舞いを COW で扱うことは出来ない．例えば，Java プログラムのエントリポイントである `main` メソッド内に次のようなコートが記述されているとき，2 行目の `noOptionCase` メソッドが本当に呼出しされるか否かはプログラム実行時にしか分からない．

```
1: if (args.length == 0)
2:     noOptionCase ();
```

COW で表現し得るのは，1 行目から 2 行目へ制御フローが到達する可能性があるか否かである．

COW の解析は静的な範囲に限られる．動的なプログラムスライシングを用いれば実行時の解析も可能ではあるが，COW はコンパイル時に織り込みの誤りを発見することを目指している．

5.4 契約を記述するプログラマ

COW における契約の記述は，プログラマが行なうべきものである．しかし一般に，ソフトウェアは複数のプログラマによって構築される．この際，どのプログラマが契約を記述するかが問題になること

が予想できる．織り込みに対する制約条件を記述するためには，メソッドの織り込み後の振る舞いに注目する必要がある．すなわち契約を記述するプログラマは，織り込み後のメソッドの振る舞いを知っていなければならない．契約を記述するためには，クラスとアスペクトの合成のされ方を知っているプログラマが行なう必要がある．

Prolog のような論理型プログラミングを知っているプログラマにとっては，COW の記述は複雑ではない．しかしそれ以外のプログラマにとっては，COW の契約記述が容易ではないかもしれない．COW では，この問題を解消するために述語のライブラリ化をサポートしている．図 4 の Updating のような契約が，述語ライブラリである．我々は様々な述語を COW の標準ライブラリとして提供することを検討しており，これにより契約記述が簡素化されるものと思われる．

5.5 検証効率

COW の契約には述語論理が用いられ，契約検証は述語論理の充足可能性によって判定される．論理式の充足可能性問題は一般に NP 完全であるが，COW では述語論理の範囲を Horn 論理に限定している．Horn 論理の充足可能性問題は多項式時間で計算可能であり，COW による検証効率は実用上問題とならない．

6 関連研究

AOP における織り込みの影響については，いくつかの先行研究が存在する [6][7][16][21]．また，AOP 言語へプログラムスライシングを適用する試みもある [4][5][20]．

ここでは本稿との関連が深い G.Kiczales らの Aspect-aware interface[14]，M.Rinard らのアドバイスの影響分類 [19]，J.Aldrich らの Open Module[1]，K.Sullivan らの Crosscutting Programming Interface[22] の 4 つを取り上げ，COW と比較する．

6.1 Aspect-aware interface

Aspect-aware interface(以降 AAIF) とは，クラスとアスペクト間の織り込みに関する情報をモジュールのインターフェースに含めたものである．AAIF を用いると，メソッドがどのアスペクトのアドバイスから織り込みを受けているかが表現できる．織り込みの前後で AAIF が変わらない限り，織り込みはプログラマの意図通り行なわれていると見なす．

これに対し COW は，織り込みによるクラスとアスペクト間の制御フローや依存関係の変化を検証する．すなわち COW が注目するのは，織り込みによるメソッドの振る舞いの変化である．これにより，COW では AAIF よりも豊富な影響解析を行なうことが出来る．

6.2 アドバイスの影響分類

M.Rinard らは，アドバイスがメソッドに織り込まれたときにメソッドが受ける影響の危険性を分類している．この分類は定性的なものであるが有用である．例えば，織り込み後にメソッドとアドバイスが同じクラスフィールドへ書き込みを行なっている場合を，彼らは干渉 (interference) と呼んでいる．

COW では基本述語の組み合わせによって，彼らが行なった分類を再現できる．例えば干渉は次のようになる．

```
/* m:method,  a:advice */
define interference(m, a) {
    statement(m, ms) &&
    statement(a, as) &&
    write(ms, f) &&
    write(as, f)
}
```

さらに COW では，定性的な分類では扱うことが難しかった織り込みの影響を表現できるようになった．例えば彼らは，織り込みを受けたメソッドがその影響で実行されなくなるか否かについて言及しているが，織り込みを受けたメソッド内で他のメソッドがどのように呼び出されるかについては言及していない．COW では図 4 に示した `Updating.plurally` 述語のように，そのような状態も表現できる．

6.3 Open Module

Open Module とは，クラスが織り込みによって影響を受けても良いジョインポイントをアスペクト側に公開するというメカニズムである．クラス側でアドバイスの影響箇所を限定することによって，織り込みでクラスの振る舞いが損なわれないことを保証できる．しかし Open Module ではジョインポイントを隠蔽できてしまうため，横断的関心事の織り込みを難しくするという副作用がある．例えば，クラ

表 2: COW と関連研究との比較

	解析対象	検証サポート
AAIF	織り込み元アドバイスの種類 (before，after 等)	IDE 上での織り込み箇所の表示
M.Rinard らの分類	アドバイスの定性的な影響分類	自動分類システム
Open Module	隠蔽されたジョインポイントへの織り込み禁止	専用の言語処理系
XPI	設計ルール	AspectJ による部分的なサポート
COW	織り込みによるメソッドの振る舞い変化	契約検証器

スフィールドに何らかのフラグを使用している場合，Open Module ではフラグの設定を行なうジョインポイントを隠蔽できる．この場合，フラグ値の変更をロギングするアドバイスは織り込み出来ない．

COW ではジョインポイントへの織り込みを制限しない．しかし述語論理によって，織り込み前後のメソッドの振る舞いは制約できる．COW は織り込みの影響を制約でき，かつ横断的関心事の織り込みも困難にしない．

6.4 Crosscutting Programming Interface

Crosscutting Programming Interface(以降 XPI) は，クラスとアスペクトを設計するためのルールを記述する方法である．設計ルール (Design Rules) の考え方は，C.Baldwin らの研究 [3] で提案されている．XPI の一部は契約と呼ばれており，AspectJ の言語メカニズムによって検証をサポートしている．AspectJ には制御フローを扱うポイントカットが提供されており，それを用いることによってプログラムの静的解析を実現できる．

しかしながら AspectJ の言語メカニズムは横断的関心事をモジュール化するために設計されており，本来プログラム解析を行なうためのものではない．契約検証を AOP 言語メカニズムと分離することによって，検証に特化した契約概念の構築が成されるべきである．

COW では契約記述のために，制御フローに加えて AspectJ には組み込まれていない制御やデータの依存関係を扱うことが出来る．しかし，我々は将来 COW の基本述語セットを拡張する必要があると考えている．具体的には，基本述語セットを AspectJ のポイントカットの表現力をカバーするよう拡張することを検討している．述語論理を用いてポイントカットを記述する研究 [9] は K.Gybels らが行なっており，本稿はこれらの研究とも関連する．

6.5 比較

表 2 は 4 つの関連研究と COW を比較したものである．この表から分かるように，COW では織り込みによるメソッドの振る舞いの変化を扱うことができるようになった．さらに，コンパイル時の自動的な契約検証もサポートされている．

7 まとめ

本稿では，AOP における契約を記述するプログラミング言語 COW を提案した．COW ではプログラム内の静的な制御フローや依存関係を織り込みに対する制約として記述できる．これにより織り込みで生じるメソッドの振る舞いの変化を検出でき，契約を記述することで織り込みの誤りを軽減することが可能になった．

コンパイル時の検証のために，現状の COW では解析を静的な範囲に限っている．しかし動的なプログラムスライシング，あるいは DbC のような実行時チェックをするための基本述語セットを新たに導入することで，コンパイル時の検証に加えてプログラム実行時の契約解析も可能になるものと考えられる．

参考文献

[1] Aldrich, J.: Open Modules: Modular Reasoning about Advice, In *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] Balzarotti, D. and Monga, M.: Using Program Slicing to Analyze Aspect Oriented Composition, In *Proceedings of Foundations Of Aspect-Oriented Languages (FOAL2004), Workshop at the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pp.25-29, 2004.

[5] Balzarotti, D., and D'Ursi, A., Cavallaro, L., and Monga, M.: Slicing AspectJ Woven Code, In *Proceedings of Foundations Of Aspect-Oriented Languages (FOAL2005), Workshop at the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pp.27-32, 2004.

[6] Clifton, C. and Leavens, G.: Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning, In *Proceedings of Foundations Of Aspect-Oriented Languages (FOAL2002), Workshop at the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pp.33-44, 2002.

[7] Dantas, D. and Walker, D.: Harmless Advice, In *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, http://homepages.inf.ed.ac.uk/wadler/fool/, 2005.

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

[9] Gybels, K. and Brichau, J.: Arranging Language Features for More Robust Pattern-based Crosscuts, In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.60-69, 2003.

[10] Helm, R., Holland, I., and Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems, In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP'90)*, pp.169-180, 1990.

[11] Java Modeling Language (JML), http://www.cs.iastate.edu/ leavens/JML/.

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

[13] Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ, In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.

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

[15] Kiczales, G. and Mezini, M.: Separation of Concerns with Procedures, Annotations, Advice and Pointcuts, In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, to appear, 2005.

[16] Krishnamurthi, S., Fisler, K., and Greenberg, M.: Verifying aspect advice modularly, In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*, pp.137-146, 2004.

[17] Monteiro, P. and Fernandes, J.: Towards a catalog of aspect-oriented refactorings, In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pp.111-122, 2005.

[18] Meyer, B.: *Object-Oriented Software Construction, Second Edition*, ISE Inc., Santa Barbara Prentice Hall Professional Technical Reference, 1997.

[19] Rinard, M., Salcianu,A., and Bugrara, S.: A classification system and analysis for aspect-oriented programs, In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*, pp.147-158, 2004.

[20] Shinomi, H. and Tamai, T.: Impact Analysis of Weaving in Aspect-Oriented Programming, In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, to appear, 2005.

[21] Skotiniotis, T. and Lorenz, D.: Cona: aspects for contracts and contracts for aspects, In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2004)*, pp.196-197, 2004.

[22] 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, In *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.

[23] Tip, F.: A Survey of Program Slicing Techniques, *Journal of Programming Languages*, vol.3, no.3, pp.121-189, 1995.

[24] Weiser, M.: Program Slicing, *IEEE Transactions on Software Engineering*, SE-10, no.4, pp.352-357, 1984.