プログラムスライスを用いたアスペクト指向プログラムのデバッグ支援環境

石尾　隆[†]　楠本真二[†]　井上克郎[†]

アスペクト指向プログラミングは，複数のオブジェクトに関連した処理をオブジェクトから分離し，新しいモジュール単位「アスペクト」として記述する．アスペクトは保守性や再利用性を向上させるが，同時に，プログラムに新たな複雑さを導入する．たとえば，アスペクトが相互に干渉して動作を阻害するなど，原因の特定が難しい欠陥が発生することがある．このような問題に対して，本研究では，プログラムスライシングを用いたデバッグ支援を提案する．プログラムスライシングは，プログラムの依存関係を解析することで開発者が扱う必要があるコードを抽出する手法であり，アスペクトがプログラムに与える影響をユーザに提示する．本研究では，アスペクト指向言語 AspectJ を対象にツールの実装を行い，適用実験を行った．その結果として，アスペクト指向プログラムにおいてアスペクトが与える影響を効果的に開発者に提示できることを示した．

Debugging Support Environment for Aspect-Oriented Program Using Program Slicing

Takashi Ishio,[†] Shinji Kusumoto[†] and Katsuro Inoue [†]

Aspect-Oriented Programming (AOP) introduces new software module named aspect for encapsulating crosscutting concerns. Although a crosscutting concern is written as an aspect to improve maintainability and modularity, AOP introduces a new factor of complexity. For example, finding defects caused by an aspect, which modifies or prevents behavior of another aspect, needs much cost. In this paper, we examine a method to support such issues in developing aspect-oriented program. We propose an application of program slicing to assist debugging. Program slicing is one of the promising approaches for fault localization, which analyzes dependencies between program statements to extract a part of program. We implement a program slicing tool for AspectJ and apply it to certain programs. The experiment shows that our approach is effective for a developer to find the influence of aspects in a program.

1. まえがき

近年，プログラムの新しいモジュール化手法としてアスペクト指向プログラミングが提案され，利用されるようになってきている[1)]．従来のオブジェクト指向プログラミングにおいて，ロギングや同期処理のような複数のオブジェクトが関わる処理のことを横断要素と呼ぶ．横断要素のコードはその処理に参加するすべてのオブジェクトに分散するため，保守性を悪化させる要因となっている．アスペクト指向プログラミングは，横断要素を単一モジュールに記述するための新しい単位「アスペクト」を導入する．

アスペクトは，プログラム中の特定の実行時点に連動して実行される処理として記述される．実行時点とは，オブジェクト間でのメッセージの送受信等のイベントのことである．実行時点はオブジェクトの枠にとらわれないため，横断要素をオブジェクトから分離して記述することが可能となる．横断要素のモジュール化は，再利用性および保守性の向上につながる．

アスペクトの応用事例は数多く報告されている[2)~4)]．しかし，アスペクトの導入が，プログラムに新しい複雑さをもたらすことも知られている．アスペクトは，オブジェクトの外部からその振る舞いを変えることができる．そのため，開発者がオブジェクトの振る舞いを把握するには，オブジェクトに関連したアスペクトをすべて調べる必要がある．プログラマの予期しない時点でアスペクトが動作するといった，発見が困難な欠陥を作り込む可能性があるためである．また，アスペクトの干渉と呼ばれる，単体ではそれぞれ正しいアスペクトが相互に動作を阻害するという問題も発生している[5)]．

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

このような問題に対して，アスペクトの予期せぬ動作の可能性や，アスペクトによって発生した障害の原因調査を支援する方法が必要である．

本研究では，アスペクト指向プログラムの開発を支援するために，コールグラフによる呼び出し関係の提示と，プログラムスライシング技術[8] を用いたデバッグ支援を行う．コールグラフとは，メソッドを頂点とし，呼び出し関係を有向辺とするグラフである．これにアスペクトの頂点と辺を加えて，予期せぬアスペクトの動作や，アスペクト間の相互依存による無限ループ発生の可能性を提示する．これによって，開発者がプログラムの実行制御に関連した欠陥を検出しやすくなる．プログラムスライシングとは，プログラム内部の依存関係を解析することで，プログラマが注目すべきコードを提示する技術である．アスペクト干渉が与える影響を調べるためには，依存関係を解析し利用するプログラムスライシング，特に実行時情報を用いたDC スライス計算[10),11)] が有効であると期待できる．

統合開発環境 Eclipse[17] をベースに，コールグラフおよび DC スライスを計算するツールの実装を行い，いくつかのアスペクト指向プログラムに対して適用実験を行った．その結果，プログラムスライシングがアスペクト指向プログラミングのデバッグ支援に適していることを示した．

以降，2. ではアスペクト指向プログラミングの特徴と問題点について説明する．3. ではコールグラフを用いた，アスペクトを組み込む際の問題検出について説明する．4. ではプログラムスライシングの概要とアスペクト指向プログラミングへの拡張について説明し，5. でアスペクト指向プログラムに対するプログラムスライシングツールの実装と評価について説明する．最後に 6. でまとめと今後の課題を述べる．

2. アスペクト指向プログラミング

2.1 アスペクト指向の特徴

アスペクト指向プログラミングは，オブジェクト指向プログラミングを基に，その弱点を補うプログラミング手法である．オブジェクト指向プログラミングでは，システムに必要な個々の機能をオブジェクトが分担し，それらが相互通信し，協調することでシステムの目的を実現する．

しかし，オブジェクトという単位でシステムの機能を分担する都合上，担当するオブジェクトを一つに決められないような特性はうまく扱えない．たとえばシステムの動作を記録していくロギング，エラーが起こったときの例外処理，データベースなどにおけるトランザクションの手続きは，システム内の複数のオブジェクトが連携して実現することが多い．このような処理は横断要素と呼ばれており，横断要素に関わる複数のオブジェクトにコードが分散し，次のような事態を引き起こす．

- 横断要素の仕様が変わると，すべての関連したコードをもれなく変更しなければならない．
- 横断要素を含んだオブジェクトの再利用性を低下させる．横断要素に関連したコードを除去して再利用するか，あるいは，関連したオブジェクト群をまとめて再利用する必要がある．
- 横断要素だけを再利用することができない．もし別の場面で同じ種類の横断要素が必要であれば，その都度実装しなければならない．

これに対して，アスペクト指向プログラミングは，横断要素を分離・記述するためのモジュール単位「アスペクト」を導入する．

アスペクト指向プログラミングでは，オブジェクト指向プログラム内に含まれる実行時点 (join points) の中から，結合の基準となる集合 (pointcut) を選択し，手続きを関連付ける．関連付けられる手続きのことをアドバイス (advice) と呼ぶ．アスペクトは，通常，一つ以上のアドバイスによって構成される．

アドバイスを結合可能な実行時点は，言語処理系によって異なるが，次のようなものが使用されている．

- オブジェクトに対するメソッド呼び出し．
- オブジェクトのメソッドの実行（動的束縛の解決後）．
- オブジェクトの持つフィールドへのアクセス．
- オブジェクトでの例外の発生．

このような実行時点に対して，アドバイスはその直前，直後，あるいは本来の処理を置き換える形式で動作する．言語処理系にもよるが，このようなアドバイスは，その実行コンテキスト，たとえば実行時点が何であるか，呼び出されようとしているオブジェクト，メソッド，引数といった情報にアクセスし，適切な処理を実行することができる．

アスペクトのコード例を**図 1** に示す．ここで定義している LoggingAspect は SomeClass.doSomething というメソッド呼び出しの直前に，その呼び出し発生を記録する．ParameterValidationAspect は，どのクラスであれ doSomething という名前のメソッドに対する呼び出しの直前に引数を検査し，条件に違反した呼び出しに対して例外を発生させるアスペクトである．

これらのアスペクトは SomeClass.doSomething への呼び出しでは両方が動作するが，相互に独立して定

```
class SomeClass {

  public void foo(int x) {
    doSomething(x);
  }
  private void doSomething(int x) {
    :
  }
}

aspect LoggingAspect {
  before(): call(void SomeClass.doSomething(..)) {
    Logger.logs(thisJoinPoint);
  }
}

aspect ParameterValidationAspect {
  before(int x):
    args(x) && call(void *.doSomething(..)) {
    if ((x < 0)||(x > Constants.X_MAX_FOR_SOMETHING)) {
      throw new RuntimeException("invalid parameter!");
    }
  }
}
```

図 1 ロギングとパラメータ検査アスペクトの例
Fig. 1 Aspect examples: Logging and parameter checking

義されているため，言語処理系が任意の順番で実行する．順番が重要な場合，AspectJ では，必要に応じてアスペクトの動作優先度を定義できる．

2.2 アスペクトのもたらす複雑さ

アスペクトの用途については広く研究されている．オブジェクト指向プログラミングで用いられているデザインパターン[12] は，複数のオブジェクトがどのように連携するかを説明した設計部品である．これは，オブジェクトの横断要素の一種であると考えられるため，アスペクトとして記述することで，パターン自身が再利用可能なソフトウェア部品となる場合があることが知られている[3]．また，ソフトウェア開発時のデバッグ支援や，分散アプリケーションでのオブジェクトを横断した性質の記述など，様々な場面でその有用性が示されつつある[2),4]．しかし，アスペクトはプログラムに次のような新しい複雑さをもたらす．

(1) 同一の条件下で動作する複数のアスペクトが存在しうる．アスペクトの動作順序によって，結果が異なる場合がある．
(2) あるアスペクトの動作中に，他のアスペクトの動作条件が成立する場合がある．二つのアスペクトが相互に条件を満たすときは，無限ループを生じることもある．
(3) 動作条件の記述を誤ると，予期せぬアスペクトの動作を招く．また，ソフトウェアを拡張した際に，動作条件の変更が必要となる場合もある．

これらのうち，(1) と (2) はアスペクトの干渉と呼ばれる問題である[5]．このような干渉を検出する，あるいは予防するための研究が数多く成されている．

また，(3) に対しては，他のアスペクトの存在，動作条件を把握する必要がある．AspectJ では，統合開発環境による部分的なサポートが行われている[15]．

アスペクトの干渉や誤作動によって生じる障害の原因を見つけ出すことは非常に難しい．特に，アスペクト干渉は必要な場合もあるため，一概に禁止するというわけにはいかない．たとえば，先に登場した，引数の値の範囲をチェックするアスペクトは，他のアスペクトからのメソッド呼び出しに対しても動作しなければならない．これに対して，アスペクトの干渉を起こさないようにアスペクトの種類を制限する研究もなされている[7] が，本研究では，実際に干渉によって動作不良に陥った場合に，その原因を特定する作業を支援することを考える．

アスペクトの複雑さは，アスペクトが他のアスペクトに対して直接，あるいは間接的に連動し，影響を与えるために生じると考えられる．そこで，アスペクトをプログラムに組み込む時点でのコールグラフを用いた制御誤りの検出と，プログラムに組み込んだ後のプログラムスライシングを用いた欠陥特定の支援を提案する．

3. コールグラフによるアスペクト干渉の検出

アスペクト指向プログラミングでは，アスペクトの結合条件の誤りやアスペクト間の干渉によって，プログラマの予期せぬ時点でアドバイスが動作し，ソフトウェア故障の原因となることが多い．

特に，アスペクトによって無限ループが生じやすいことが知られている[14]．これに対して，実行時にデバッガ等で検出するのではなく，プログラムをコンパイルした段階で，その可能性を検出できることが望ましい．

無限ループが発生する可能性を最も簡単に検出するものとして，メソッドを頂点とし，メソッド間の呼び出し関係を辺とするコールグラフ (Call Graph) がある．グラフ内のある頂点 v から出発して v に到達可能な経路があるとき，無限ループの可能性がある，ということになる．

コールグラフをアスペクト指向に対して拡張するために，あるメソッド内部にアドバイスが連動する実行時点が含まれているとき，そのメソッドからアドバイスへの呼び出しがあると考え，アドバイスの頂点と動作辺を追加する．メソッド，アドバイスを頂点とし，メソッド呼び出し，アドバイスの起動を辺としたコールグラフを構築する．あるメソッド m の頂点 v_m からアドバイス adv の頂点 v_{adv} に到達可能であるとき，m は adv に影響を受けている，と考える．あるアドバイスが自分自身の影響を受けるとき，無限ループに

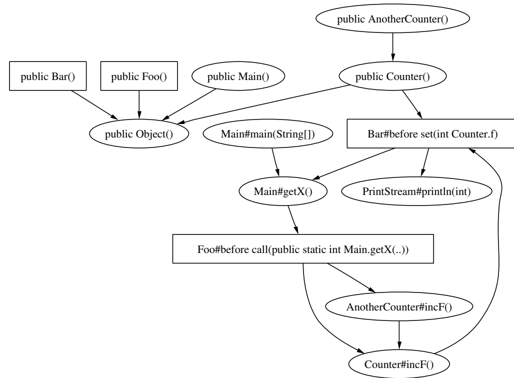

図 2 コールグラフ例
Fig. 2 A call graph

陥る可能性がある．

コールグラフは，ソースコードを解析することで容易に計算が可能である．他のアスペクトの干渉検出法として，形式的仕様記述を用いる手法などが考えられるが，それらに比べて実装が容易であり，開発者にとって直観的な情報が提示できる点が特徴である．

アスペクトの動作順序や相互依存関係を開発者が細かく指定することができる処理系[5]や，そもそも干渉を起こさないような限定された種類のアスペクトの利用[7]も研究されている．そのような場合でも，コールグラフを用いて依存関係を表示することで，予期せぬアスペクト間の依存関係の存在を開発者が確認できることは有益である．

コールグラフには，次のような弱点も存在する．まず，コールグラフは，プログラムの規模に比例して頂点数が増加していく．これに対しては，ループ部分だけを抽出し図示する，あるいは警告メッセージの出力と併用するなどして，開発者に把握しやすい形式の情報を提供する必要がある．また，コールグラフでは，開発者が意図して作成した再帰呼び出しなども出力される．これには，開発者が検出結果を効率的にフィルタリングする機構などが必要となる．

コールグラフの具体例を**図 2** に示す．楕円形の頂点はメソッド，矩形の頂点はアドバイスを示しており，内部にシグネチャを示している．このグラフでは，メソッドを表現する頂点 Main:getX からアスペクトを経由したループが存在していることから，無限ループに陥る可能性を知ることができる．

このようにコールグラフを用いて，開発者はアスペクト間の制御関係が意図したものであるかどうか確認し，無限ループの可能性を除去することができる．開発者が意図して再帰ループなどを作成する可能性もあるため，ループが検出されてもユーザが望むならプログラムをそのまま実行することも可能にしておく．

4. プログラムスライシング

プログラムスライシングは，プログラムに含まれる手続き呼び出し関係や変数の参照・代入関係などの依存関係を解析し，プログラム内の注目すべき部分だけを抽出，提示する技術である[8]．具体的には，ある文のある変数を入力として，その変数の値に影響を与える文の集合を取り出す．入力として与えるコード内の変数の参照位置をスライス基点と呼び，抽出されたプログラム文の集合をプログラムスライス，あるいは単にスライスと呼ぶ．

4.1 スライス計算のアルゴリズム

プログラムスライシングは，次の三つのフェイズからなる．

(1) プログラムからの依存関係の抽出
(2) プログラム依存グラフの構築
(3) グラフ探索によるスライスの抽出

(1) は，データ依存関係，制御依存関係の二つの依存関係を解析するフェイズである．データ依存関係とは，変数の代入と参照の関係である．プログラム中の二つの文 s, t について以下の条件が成り立つとき，s から t に対して変数 v に関するデータ依存関係が存在すると言う．

- 文 s で変数 v に値を代入している．
- 文 t で変数 v の値を参照している．
- 文 s から t に到達可能な制御フローがある．
- 文 s から t に到達する制御フローのうち，途中で変数 v への代入文がないような経路が少なくともひとつ存在する．

また，制御依存関係とは，実行制御文と制御される文の関係である．プログラム中の文 s, t について，以下の条件が成り立つとき，s から t への制御依存関係が存在すると言う．

- s が条件節である．
- t が実行されるかどうかが，s の判定結果によって決まる．

フェイズ (2) で，これらの依存関係と，手続き・メソッドの呼び出し関係を用いて，プログラム文を頂点，依存関係を有向辺としたグラフを作成する．このグラフをプログラム依存グラフ（Program Dependence Graph, 以下 PDG）と呼ぶ．フェイズ (3) では，PDG 上でスライス基点となる頂点を選び，有向辺を逆向きに探索していくことで，スライス基点となった変数に影響を与える文を抽出していく．得られた頂点集合をエディ

表 1 スライスの種別と特徴
Table 1 Type and characteristics of slicing methods

種別	制御依存	データ依存	スライスサイズ
静的	静的	静的	大 (すべての可能性)
DC	静的	動的	中 (特定の実行系列)
動的	動的	動的	小 (特定の実行系列)

タなどに表示されたソースコード上へ反映し，プログラマへ情報を提供する．

プログラムスライシングの性能は，その依存関係の抽出方法によって決まる．スライスの種別を**表 1** に示す．ソースコードから依存関係を取得する静的スライスは，すべての可能性を抽出するため，プログラム理解や検証に用いられる[8]．動的スライスはある特定の入力に対する実行系列上で依存関係を解析し，実際に依存関係が発生したコードだけを抽出するため，デバッグ等に利用される[9]．しかし，実行系列を完全に保存する必要があるため，必要なコストは非常に高い．それらの中間的な Dependence-Cache(DC) スライスは，実行系列を保存せずにデータ依存関係だけを動的に取得することで，実用的なコストで特定の実行系列に関する情報を取得できる[10),11]．

4.2 アスペクト指向プログラムに対するスライス計算の適用

アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグ支援を行う．単一の実行時点で複数のアスペクトが動作する場合の実行順序は処理系に依存するため，プログラムの実行時情報を用いる DC スライスが適切であると考える．この時点で，コールグラフを用いて無限ループの可能性は除去されており，プログラムの実行は少なくとも終了すると考えられる．そこで，開発者が何らかのテストケースを実行し，得られた結果が不正であり，その原因を調査するという状況を想定する．実行結果が不正なテストケースを実行し，その実行を観測して得られた情報を基にプログラム依存グラフの構築とスライス計算を行い，開発者にフィードバックを行う．

オブジェクト指向プログラムに対する DC スライスの例を**図 3** に示す．このプログラムは，IncrementCounter と ShiftCounter という二つのクラスを用意し，引数に応じて一方だけを用いて，結果を出力するプログラムである．ここで，IncrementCounter を用いて動作するような入力 “inc” を与えたときの実行に対して，スライス基点として，最終出力である矩形 (d) の変数を選択すると，矩形 $(a), \cdots, (f)$ で囲まれた部分が DC スライスとなる．

プログラムの出力が不正な値となった場合，開発者は

```
class Count {

  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("java Main [sft|inc]");
      return;
    }

    Counter counter;
    boolean isIncrementCounter = false;                 (a)
    if (args[0].equals("inc")) {
      counter = new IncrementCounter();
      isIncrementCounter = true;
    } else if (args[0].equals("sft")) {
      counter = new ShiftCounter();
    } else return;

    int x = 0;
    for (int i=0; i<1000; ++i) {                         (b)
      counter.proceed();
      x = counter.value();
      if (x > 1000) break;
      System.out.println(x);
    }

    String result;
    if (isIncrementCounter) {                           (c)
      result = "increment counter = ";
      result = result + Integer.toString(x);
    } else {
      result = "shift counter = ";
      result = result + Integer.toString(x);
    }
    System.out.println(result);   (d)
  }
}

abstract class Counter {
  private int count = 1;
  public Counter() {}                                    (e)
  public int value() { return count; }
  public void proceed() { count = newValue(count); }
  abstract protected int newValue(int old);
}

class IncrementCounter extends Counter {
  protected int newValue(int old) {
    return old + 1;   (f)
  }
}

class ShiftCounter extends Counter {
  protected int newValue(int old) {
    return old << 1;
  }
}
```

図 3 DC スライス計算例
Fig. 3 DC slice example

スライスに含まれた部分だけを調べればよく，開発者が欠陥の原因を特定する作業を軽減することができる．

現在，オブジェクト指向言語 Java に対するプログラムスライシングが既に実現されているが，アスペクト指向プログラムに対する適用は基本的な手段の提案のみであり，詳しい議論はなされていない[16]．

本研究では，対象とするアスペクト指向言語として AspectJ を選択し，Java に対するプログラムスライシングを拡張することで実現する．具体的には，コールグラフと同様に，アドバイスが連動する実行時点からその動作するアドバイスへと手続き呼び出しと同様の関係があると考え，アドバイスがアクセスするプログラム情報はメソッドの引数としてデータが渡される場合と同様に考え，その依存関係を PDG に追加する．

4.3 プログラム実行時情報の解析

DC スライスでは，プログラムの実行時情報が必要

となる．実行時情報を収集する処理は，それ自体が横断要素の一つであると考えられる．そこで，データ収集処理をアスペクトとして記述し，対象プログラムに組み込む．ここで必要なプログラム実行時情報とは，次の通りである．

動的データ依存関係 どこで代入された変数がどこで用いられたか，という依存関係である．計算手順を次に示す．

事前準備 使用される各変数 v に対応したキャッシュ $C(v)$ を用意する．ただし，オブジェクトのフィールドに関しては，インスタンスごとにキャッシュを用意する．

文 s で v に値が代入された時点 $C(v)$ に s を記録する．

文 t で v の値が参照された時点 $C(v)$ に保存されている代入文 s から参照文 t への v に関するデータ依存関係を抽出する．

動的束縛の解決 あるメソッド呼び出し文に対して，実際に動作したメソッドがどれか，という情報である．静的解析に比べてプログラムのコードを絞り込むための有力な情報である．計算手順を次に示す．

メソッド呼び出し直前 メソッド呼び出し文の位置と，呼び出そうとしているメソッドを記憶する．

メソッド実行直前 (動的束縛の解決後) 呼び出し文の位置から，呼び出されたメソッドへのメソッド呼び出し情報を記録する．

我々は既に実行時情報収集アスペクトを，Java を対象として実現している[2]．AspectJ ではローカル変数の監視はできないので，実行時のオブジェクトの個々のインスタンスの識別と，フィールドのデータ依存関係を取り出すだけでも，十分に有効な結果であることは Java を対象としたプログラムスライシングの実現で既に示されており[2]，この手法を AspectJ に対して拡張して利用する．このアスペクトを加えた際の干渉の発生については，コールグラフを用いた制御エラーの検出と，AspectJ でのアスペクト間の動作優先度の定義を用いて対処する．

5. 実装と評価

5.1 実装の概要

デバッグ作業は，コードを書き換え，テストケースを再実行するという反復作業として考えられる．このような反復テストをサポートする統合開発環境も多く，プログラムスライシングのような支援ツールも，それらと連携して利用可能であることが望ましい．本研究では，統合開発環境として Eclipse[17] を選択し，これに統合する形式でツールの実装を行った．

Eclipse はオープンソースの統合開発環境で，Java で記述したプラグインを追加することでエディタやコンパイラの機能を拡張することができる．Java および AspectJ を対象としたソースコード入力支援等のプラグインが既に開発されているため，それらを利用してスライス計算機能を実装した．

Eclipse のプラグインでは，ファイルの保存やコンパイルの終了など，重要なイベントに応じて動作するアクションを定義することができる．そこで，コンパイル時に静的な依存関係の抽出，コールグラフの構築を行うものとした．コールグラフ内部にループなどが検出された場合には，それをメッセージとしてユーザに通知する．また，開発者はコンパイルした時点で静的スライスを計算することができ，実行時情報が存在するときは，それを用いて DC スライス計算を行う．また，エディタ上でソースコードの選択を行い，その部分をスライス基点としてスライス計算を実行できるようにした．

ソースコードの規模としては，ユーザインタフェースおよびスライス計算部が 4300 行，動的解析部が約 1000 行となった．構文解析等には，既存のプラグインの機能を利用している．

表 2 適用対象
Table 2 Target codes

名称	サイズ (LOC)
ChainOfResponsibility	517
Observer	667
Singleton	375
Mediator	401
Strategy	465

5.2 適 用 実 験

作成したツールを，Java および AspectJ を用いたデザインパターンの実装例としてインターネット上で公開されているコード[13] に対して適用した．ここでは，デザインパターンのうち，**表 2** に示すような，アスペクトとして記述することが効果的とされている 5 種類のパターン[3] の実装を用いた．これらのコードをそれぞれコンパイルして実行し，スライス計算を行った．以降，5.3. ではスライス結果について，5.4. では計算コストについて述べる．

5.3 スライス結果の評価

スライス基点としてアスペクトの中の頂点を選んで，スライス計算を行った．計算した結果のうち，Singleton パターンと Strategy パターンの実装例に対する

表 3 スライスサイズ
Table 3 Size of a static and DC slice

対象	静的スライス	DC スライス
Singleton	200	173
Strategy	113	60

```
class Sample {

  private int aField;

  public int foo() {
    int x = bar();
    :
    :
  }
                                        activate
  protected int bar() {
    return 0;  // never executed
  }

  private int baz() {
    return aField;                  call
  }

}

aspect redirectMethodCall {
  int around(Sample sample):
    this(sample) && call(int Sample.bar()) {
    return sample.baz();
  }
}
```

図 4 メソッド置換アスペクトのスライス結果
Fig. 4 A slice including an aspect which replaces a method call

静的スライスと DC スライスのサイズの例を示す．これらは，静的スライスがすべての可能性を抽出する性質によって生じる差であった．他のパターンについては，動的なデータ構造を経由した依存関係を含んでおり，単純な静的スライスでは抽出することができなかった．静的スライスが正しい結果を出力できなかったため，比較の対象から除外している．

得られたスライス結果と同様の結果を手作業で得るためには，複数のファイルに分散して定義されたクラスやアスペクトの定義を順次追跡する必要がある．プログラムスライシングを用いて必要な箇所を提示することで，作業効率の向上が期待できる．

また，プログラムスライシングを用いることで，開発者の誤解しやすい点を指摘することができる．**図 4** は，あるメソッド呼び出しを別のメソッド呼び出しに置き換えるアスペクトを含んだプログラムに対するスライス結果である．このようなメソッド置き換えアスペクトは，単体テスト時の便宜的な実装の変更や，未実装部分の補完などに使用される．通常，アスペクトがクラスとは別のファイルに記述されているため，このようなアドバイスの存在による依存関係の変化を認識することは難しい．図 4 の例では一行分の違いしかないが，実際にはこのクラスを継承したクラスや，そこで使用されている他のクラス，アスペクトが関連するため，誤認の影響は大きな範囲に及ぶ可能性がある．このような，アスペクトによって変化した振る舞い，依存関係を指摘することで開発者を支援できることが，プログラムスライシングの大きな利点である．

表 4 動的解析処理の実行時間
Table 4 Time cost of dynamic analysis

対象	通常実行	動的解析付き実行
ChainOfResponsibility	3.76	3.93
Observer	0.32	0.37
Mediator	3.21	5.69
Singleton	0.14	0.32
Strategy	0.18	0.22

5.4 実行コストの評価

スライス計算の実行プロセスにおいて，実行コストは以下の場面で影響を与える．

コンパイル時 静的情報収集のために計算コストとメモリを必要とする．

動的情報解析時 動的情報収集モジュールを付加するため，通常の実行に比べて計算コストとメモリを必要とする．

スライス計算実行時 スライス計算は，構築されたグラフの探索問題であり，頂点数に比例した時間で終了する．

コンパイル時の処理は，AspectJ コンパイラが意味解析なども含めて構築したモデル情報にアクセスできるため，構文木を 1 回トラバースするだけで終了する．アルゴリズムの計算量の厳密な評価ではないが，トラバース処理は構文木のサイズに比例し，コンパイラの計算量に比べれば相対的に小さい．動的情報解析に必要な時間コストを**表 4** に示す．

また，メモリ消費量に関しては，主にプログラムの規模によって影響を受けるが，アスペクトの種類によっても大きく異なる．たとえば，デザインパターンを実装したアスペクトを含んだ約 10000 行のコードに必要な解析コストは約 20MB だったが，このプログラムのすべてのメソッド呼び出しと実行，フィールドのアクセスに対して動作するアスペクトを加えると，1 割に相当する約 1000 行程度のアスペクトであったにも関わらず，解析に必要なメモリは 5 倍の約 100MB に増加した．これは，すべてのメソッド呼び出しという動作条件がプログラム中の各所で成立するため，他のアスペクトの中でさえも動作するため，アスペクトの呼び出しを表現する辺の数が急激に増加したことが原因であると考えられる．

このような解析に必要なメモリの増加の問題は，As-

pectJ のコンパイラ開発者らによって指摘されている[14]もので，スケーラビリティの実現への課題となっている．

6. む す び

本研究では，アスペクト指向プログラムに対してプログラムスライシングを適用し，デバッグ支援を行うための開発環境の実装を行った．

アスペクト指向プログラミングの特徴は，横断要素と呼ばれる複数のオブジェクトが関わる処理を，アスペクトという単位でモジュール化することにある．横断要素のコードが複数のオブジェクトに分散することがなくなり，保守性，再利用性を改善することができる．

アスペクトは，行うべき処理と動作条件をペアにしたアドバイスという単位の集合として記述される．アドバイスは，オブジェクト側のコードを書き換えることなく，オブジェクトが行う処理を追加あるいは変更する．そのため，予想外のアスペクトの動作などにより，発見の困難な欠陥を作りこむ可能性がある．

これに対して，まず，コールグラフを用いて，無限ループなど，実行を不可能にするような制御誤りを取り除く．プログラムの出力が得られるようになった段階で，出力結果が正しく得られない場合に，プログラムスライシングを用いた原因の調査を行う．

アスペクト指向プログラムに対するプログラムスライシングを，アスペクトの起動をメソッド呼び出しの一種とみなすことで実現した．特定のテストケースに対するデバッグ支援を重視するため，スライス計算アルゴリズムには，部分的に実行時情報を用いる DC スライスを採用した．

コールグラフ構築および DC スライス計算ツールを統合開発環境 Eclipse へのプラグインの形式で実装し，アスペクトを用いたサンプルプログラムを対象に評価実験を行った．その結果，プログラマが誤解しやすいアスペクトの依存関係などを効果的に提示できることを確認した．

今後は，プログラムスライスがデバッグ支援に与える効果についての評価を行っていく予定である．また，アスペクト指向プログラムに対するプログラムスライシングでは，アスペクトの定義によってメモリ消費量が大きく左右されることから，大規模なプログラム，多数のアスペクトを扱うためのスケーラビリティの改善を今後の課題とする．

参 考 文 献

1) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin: "Aspect Oriented Programming", Proceedings of ECOOP, vol.1241 of LNCS, pp.220-242(1997).
2) 石尾隆, 楠本真二, 井上克郎: "アスペクト指向プログラミングのプログラムスライス計算への応用", 情報処理学会論文誌, Vol.44, No.7 (採録決定).
3) J. Hannemann, G. Kiczales: "Design Pattern Implementation in Java and AspectJ", Proceedings of OOPSLA 2002, pp.161-173, Nov. (2002).
4) S. Soares, E. Laureano, P.Borba: "Implementing Distribution and Persistence Aspects with AspectJ", Proceedings of OOPSLA 2002, pp.174-190, Nov. (2002).
5) R. Pawlak, L. Seinturier, L. Duchien, G. Florin: "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", Proceedings of REFLECTION 2001, pp.1-24 (2001).
6) I. Kiselev: "Aspect-Oriented Programming with AspectJ", Sams Publishing, Indiana (2002).
7) 一杉裕志, 田中哲, 渡部卓雄: "安全に結合可能なアスペクトを提供するためのルール", ソフトウェア科学会第 19 回大会, Sep.(2002).
8) M. Weiser: "Program slicing", IEEE Transactions on Software Engineering, SE-10(4):352-357(1984).
9) H. Agrawal and J. Horgan: "Dynamic Program Slicing", SIGPLAN Notices, Vol.25, No.6, pp.246-256 (1990).
10) T. Takada, F. Ohata, K. Inoue: "Dependence-Cache Slicing:A Program Slicing Method Using Lightweight Dynamic Information", Proceedings of IWPC2002, pp.169-177, June (2002).
11) F. Ohata, K. Hirose, M. Fujii, and K. Inoue: "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", Proceedings of APSEC2001, pp.273-280(2001).
12) E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley (1995).
13) Jan Hannemann: "Aspect-Oriented Design Pattern Implementations", http://www.cs.ubc.ca/ jan/AODPs/
14) AspectJ Team, "The AspectJ Programming Guide", http://aspectj.org/doc/dist/progguide/
15) AspectJ Team, "AspectJ Development Environment", http://www.eclipse.org/ajdt/
16) J.Zhao, "Slicing Aspect-Oriented Software", Proceedings of IWPC2002, pp.251-260 (2002).
17) Eclipse Project, http://www.eclipse.org/