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

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，ロギングや同期処理のような複数のクラスを横断した処理を扱うために「アスペクト」という新しいモジュール単位を導入したプログラミング手法である．従来のオブジェクト指向プログラムでは複数のオブジェクトに分散していたコードを，単一のアスペクトに簡潔にまとめることができ，保守性や理解容易性を向上させることが可能である．この応用として，プログラム解析に用いるプログラムの実行時情報の収集がある．実行時情報の収集は，システム全体のコードに影響を与えるため，単一のモジュールとしてプログラムに組み込む，ということは考えられていなかった．本論文では，Java におけるプログラムスライス計算を行うための動的情報収集モジュールを AspectJ を用いて記述し，その利便性と実現コストの軽減について考察する．

Application of Aspect-Oriented Programming to Calculation of Program Slice

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

Aspect-Oriented Programming (AOP) introduces new software module named aspect for encapsulating crosscutting concerns, such as logging, synchronization, etc. Such concern might be distributed among objects in Object-Oriented Programming, but it can be written in single aspect. One useful application of AOP is to modularize collecting program's dynamic information for program analysis. Since collection of dynamic infomation affects over all target program, nobody built this functionality as one module into target program. In this paper, we develop program sliging system using AspectJ, and describe benefits, usability, cost effectiveness of the module of dynamic analysis.

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)]．アスペクト指向プログラミングの特徴は，ロギングや同期処理のような複数のクラスを横断した処理をモジュール化する新しいモジュール単位「アスペクト」を導入していることにある．従来のオブジェクト指向プログラミングでは，複数のオブジェクトを横断した処理は，当然ながら単一のオブジェクトにカプセル化することができない．アスペクト指向では，このような処理を単一のアスペクトというモジュールで記述し，コードが複数のオブジェクトに分散することを避けることができる．

一方で，アスペクト指向の考え方の応用事例というのはあまり多く報告されていない．アスペクト指向の考え方が適当であるオブジェクトを横断した処理の 1 つとして，プログラムの動的情報の収集が考えられる．プログラムの動的情報とは，簡単にいうと，ある入力が与えられたときにプログラム中で実行された命令の系列である．プログラムの動的情報の収集は，プログラムスライスの計算[2)] やプログラム実行時の動的な複雑さの計算[9)] において特に必要とされている．

プログラムスライシングは，Weiser[16)] によって提案されたものである．プログラムソース中のある地点のある変数の値に影響を与える，つまりその変数に依存関係を持つような文の集合を抽出する技術で，保守やデバッグに有効な手法である．

近年のソフトウェア開発環境においては，Java や C++などのオブジェクト指向言語が頻繁に利用されるようになってきている．オブジェクト指向言語では，クラスや継承などオブジェクト指向独特の概念が導入されており，数多くの実行時決定要素が含まれている．このようなプログラムに対するスライス計算では，プログラムを実行した際にその経過を観測し，実際に実行されたプログラムの情報をスライスの結果に反映させることが有効である．プログラムスライシング手

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

法の 1 つである Dependence Cache（DC）スライスは，プログラムの制御構造については静的に解析するが，データ依存関係は実行時に解析する方法で，低コストで十分正確なスライスを得られることが知られている[2]．

オブジェクト指向プログラミング言語である Java を対象とした DC スライス計算では，動的データ依存関係の解析の実現が重要な課題となっている．動的データ依存解析は，解析対象のプログラムを実際に実行している経過を観測し，システムに含まれるオブジェクトを横断してデータの依存関係を追跡していく処理である．この処理は，オブジェクト指向では単一のモジュールとして記述することができず，プリプロセッサによるソースコードの変換[11]，Java Virtual Machine（JVM）の改造[3] という形で実現されていた．しかし，前者は構文上の変換規則を記述することが困難であり，後者は特定の JVM の実装に依存した実現になるという問題があった．

本論文ではこのような問題に対して，アスペクト指向プログラミングを導入することで，動的データ依存解析をアスペクトによって記述し，DC スライスを効率良く算出する手法について提案する．具体的には，AspectJ [13] を用いて動的データ収集のモジュールを記述し，JVM 改造によるアプローチとの比較実験を行った．この結果，アスペクト指向プログラミングによるアプローチが，従来の手法と比較して，若干の正確性を犠牲に，大幅なコストの改善が行えることを確認した．

以降，2 章ではプログラムスライスについての概要を，3 章ではアスペクト指向プログラミングについての概要と動的データ依存関係のアスペクトによる実現方法を説明する．4 章で評価実験とその結果について説明し，最後に，5 章にまとめと今後の課題を述べる．

2. プログラムスライス

プログラムの実行時情報解析が有効な技術の 1 つとして，プログラムスライシング（Program Slicing）技術がある．

プログラムスライシング技術とは，プログラム中のある文 s におけるある変数 v（**スライス基点** $\langle s, v\rangle$ と呼ぶ）に対して v の値に影響を与えるすべての文をプログラムから抽出する技術で，その結果取り出された文の集合をプログラムスライスまたは単にスライス（Slice）と呼ぶ．v に影響を与える文を抽出することで，プログラム中に存在するフォールトの位置特定に有効であるだけでなく，プログラム保守，プログラム理解などにも利用される．

スライスの計算には様々な手法が存在するが，本研究では，プログラム依存グラフによるスライス計算手法を用いる[7]．

2.1 プログラム依存グラフ

プログラム依存グラフ（*Program Dependence Graph*，以降，PDG）は，プログラム中の依存関係を表現する有向グラフである．PDG の節点はプログラムに含まれる条件判定，代入文，入出力文，手続き呼び出し文を表し，その有向辺は 2 つの節点間の制御依存関係およびデータ依存関係を表す（それぞれを制御依存辺，データ依存辺と呼ぶ）．また，関数間にわたるデータ依存関係を表現するために特殊節点および特殊辺も存在する[8]．

制御依存関係，データ依存関係は，それぞれ次のように定義される．

制御依存関係： プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に**制御依存関係**（*Control Dependence*，CD 関係）が存在するという．

（1） s は条件文である．

（2） t が実行されるかどうかは，s の判定結果に依存する．

データ依存関係： プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に変数 v に関する**データ依存関係**（*Data Dependence*，DD 関係）が存在するという．

（1） s で v が定義される．

（2） t で v が参照される．

（3） s から t へ，途中で変数 v を再定義している文が存在しないような経路が少なくとも 1 つ存在する．

スライス基点 $\langle s, v\rangle$ に対するプログラムスライスは，依存関係解析によって PDG を構築した後，s に対応した PDG の節点 V_s から，逆方向に制御依存辺およびデータ依存辺を経て推移的に到達可能な節点集合に対応する文の集合を計算することで得られる．

制御依存関係については，ソースコードから解析するだけでも十分な情報を得ることができる．しかし，オブジェクト指向言語で記述されたプログラムには，オブジェクトの多態性や例外処理のような実行時決定要素が数多く含まれる．データ依存関係をソースコードから解析する場合，実行される可能性のあるすべての経路を考慮する必要があり，解析結果の正確性が低下する．デバッグやプログラム理解にプログラムスライシング技術を用いる場合，特定の入力に対するプロ

```
1: a[0] = 0;
2: a[1] = 1;
3: a[2] = 2;
4: a[3] = 2;
5: a[4] = 2;
6: read(c);
7: b = a[c] + 5;
```

図 1 配列を含むプログラム
Fig. 1 Example program using array.

グラムの動作を，より正確に解析したいという要求がある．このような要求に対して，Dependence Cache（DC）スライスが提案されている[11]．

DC スライスは，実際にプログラムを実行してデータ依存関係解析を行い，実行時決定要素の情報を収集する．一方で，制御依存関係については静的に解析を行うため，実行系列を保存する必要はなく，解析コストを低く抑えることができる．

2.2 DC スライスにおける動的データ依存関係解析

プログラム中のある文 s においてある変数 v が参照されるとき，v を定義した文 t が分かれば，s から t の間に，v に関するデータ依存関係が存在することが把握できる．つまり，各変数 v について，その変数がどこで定義されたかを保存しながらプログラムを実行すれば，動的なデータ依存関係解析を実現することができる．

そこで，DC スライスの計算では，プログラム中で用いられるすべての変数 v に対し**キャッシュ**（*Cache*）$C(v)$ を用意する．$C(v)$ に変数 v が最後に定義された文番号を格納しておき，文 t の実行時に変数 v に対するアクセスがあった場合，次のような処理を行う．

文 t で v が定義された場合

$C(v)$ の値を t の文番号に更新する．

文 t で v が参照された場合

$C(v)$ に対応する命令と t に対応する命令の間に発生する v に関するデータ依存関係を抽出する．

例として，**図 1** のような配列を含むプログラムに対して動的データ依存関係解析を行う場合を考える．入力として変数 c に 0 を与えて実行させたときの各実行時点における各変数 v のキャッシュ $C(v)$ の推移を**表 1** に示す．

文 1 から文 6 では，それぞれ変数 $a[0]$，$a[1]$，$a[2]$，$a[3]$，$a[4]$，c が定義されているため，文 6 の実行が終了した時点で $C(a[0]) = 1$，$C(a[1]) = 2$，$C(a[2]) = 3$，$C(a[3]) = 4$，$C(a[4]) = 5$，$C(c) = 6$ となる．文 7 で変数 $a[0]$ が参照されるため，文 7 の実行時に文 $C(a[0])$，つまり文 1 と文 7 の間に $a[0]$ に関するデータ依存関係が発生することになる．

表 1 図 1 におけるキャッシュの推移
Table 1 Cache transition of Fig. 1.

実行文	$C(a[0])$	$C(a[1])$	$C(a[2])$	$C(a[3])$	$C(a[4])$	$C(b)$	$C(c)$
1	1	-	-	-	-	-	-
2	1	2	-	-	-	-	-
3	1	2	3	-	-	-	-
4	1	2	3	4	-	-	-
5	1	2	3	4	5	-	-
6	1	2	3	4	5	-	6
7	1	2	3	4	5	7	6

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

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

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

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

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 2 ソースプログラムと入力 “inc” に対する，基点 (d) に関する DC スライス
Fig. 2 Source program and DC slice example (slice criteria (d), input = “*inc*”).

上述のようにして動的に抽出したデータ依存関係と，静的に抽出される制御依存関係を用いて PDG を構築する．そして，スライス基点に対応する節点からグラフを探索し，到達可能な節点集合を求め，それに対応する文を得ることによって DC スライスが計算される．

DC スライスの例を，**図 2** に示す．この Java で記

述されたソースコードに対し，引数 “*inc*” を与えて実行し，スライス基点として矩形 (d) に含まれる変数 *result* に関する DC スライスを計算すると，矩形 (a)〜(f) で囲まれた部分が DC スライスの結果となる．

3. アスペクト指向によるプログラム動的情報の収集

DC スライスの計算には，プログラムの実行時情報が必要である．実行時情報を収集する方法については様々な実現方法があるが，実装に必要なコスト，あるいは実行時のコストが高くつくものが多い．本研究では，アスペクト指向プログラミングを用いることで，それらのコストの軽減を試みる．

3.1 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では解決できない横断要素の分離を実現することを目標としている．

オブジェクト指向言語では，通常，オブジェクトという単位によってソフトウェアを分解，モデル化する．しかし，ロギングや同期処理といった，複数のオブジェクトを横断する処理は，単一のオブジェクトにカプセル化することができない．したがって，このような処理を行うコードは複数のオブジェクトに分散するが，分散したコードの一貫性の維持は非常に難しく，プログラムの保守性や再利用性を悪化させる．

アスペクト指向プログラミングは，そのような横断要素を分離，記述するためのアスペクトという新たなモジュール単位を導入する．分離されたアスペクトは，オブジェクト指向で記述されたプログラムに Aspect Weaver と呼ばれる処理系によって結合される．

アスペクトの結合は，任意の場所に行われるわけではなく，プログラムの特定の実行時点（Join Point）で行われる．開発者は，Join Point の中から必要な部分を pointcut と呼ばれる集合として取り出し，横断処理をそれに連動して動作する処理 advice として記述する．Aspect Weaver は，分離して記述された処理を，プログラム中の pointcut に埋め込み，実行可能なプログラムを生成する．

Java に対する Aspect Weaver の 1 つである AspectJ では，**表 2** に示す pointcut 指定子を用いて Join Point を選択する．これらに対して，before（直前），after（直後），around（前後）の 3 種類の形式で，advice を結合することができる．

AspectJ は，ソースコードレベルで処理を行う．Java コードとアスペクトコードを入力として受け取り，それらを結合した Java ソースを中間的に生成する．このとき，そのアスペクトがソースコード内のどこに結合されたかという位置情報が得られるため，AspectJ はこの情報をコードに埋め込み，アスペクトから参照できるようにする機能を提供している．アスペクトは，たとえば，呼び出されたメソッドがどのクラスに属するかだけでなく，どのファイルの何行目に位置しているか，という情報まで扱うことができる．

表 2 AspectJ で利用可能な pointcut 指定子
Table 2 Pointcut designators (AspectJ).

Join Point	意味
call	メソッド，コンストラクタの呼び出し
execute	メソッド，コンストラクタの実行
get	フィールドの参照
set	フィールドへの代入
handler	例外処理の実行

```
aspect LoggingAspect {

  pointcut AllMethodCalls():
    !within(LoggingAspect) &&
    call(* *.*(..));

  pointcut MethodExecs():
    !within(LoggingAspect) &&
    execution(* somepackage.*.*(..));

  static Stack callStack = new Stack();
  static JoinPoint lastCall = null;

  Object around(): AllMethodCalls() {
    callStack.push(thisJoinPoint);
    lastCall = thisJoinPoint;
    proceed(); // execute original call
    lastCall = callStack.pop();
  }

  before(): MethodExecs() {
    if (lastCall != null) {
      Logger.logs("executed",
        lastCall.getSignature(),
        lastCall.getSourceLocation(),
        thisJoinPoint.getSignature(),
        thisJoinPoint.getSourceLocation());
    }
  }
}
```

図 3 動的束縛の記録を行うアスペクト
Fig. 3 Aspect which records dynamic bindings.

3.2 アスペクトの具体例

AspectJ のコード例を**図 3** に示す．このコードは，動的束縛が実行時にどのように解決されたかを記録するものである．具体的には，プログラム実行中のメソッド呼び出しを監視し，呼び出しに対して実際に実行されたメソッドの情報を記録している．

AspectJ を用いない場合，この処理を実現するためには，対象となるクラスのすべてのメソッドの実装の

先頭と末尾，およびメソッド呼び出しの直前と直後にコードを記述しなければならない．しかし，AspectJでは，複数のクラスやメソッドの名前をパターンマッチさせる記号 “*” を用いて指定することで，アスペクトのコードを非常に小さく，簡潔に記述することができる．クラス側でのメソッドの追加や削除に対してコードの変更が不要になるだけでなく，アスペクトをクラスとは独立して再利用することが可能となる．

3.3 プログラムの動的解析

プログラムの動的情報の解析は，プログラムスライス計算やプログラムの動的複雑度の計測などに必要とされる技術である．

従来，オブジェクト指向言語 Java を対象としたプログラム実行時情報の解析には，次のような実現方法が利用されていた．

(a) プリプロセッサによる解析命令の埋め込み[11)]

(b) Java Virtual Machine Profiler Interface（JVMPI）の利用[12)]

(c) Java Debugger Interface[14)] の利用☆

(d) Java Virtual Machine（JVM）の改造[3)]

(a) は，Java の構文木上での変換ルールを作成し，解析命令を埋め込む方法である．しかし，解析命令は，マルチスレッド動作への対応や例外処理など，数多くの要素に対処する必要があるため，プリプロセッサの構文的な変換だけでは対応することが難しい．またプリプロセッサそのものの保守性や再利用性，他のプリプロセッサとの競合への対策なども必要であり，実現コストが高くなる傾向にある．

(b) は，JVM に用意されているプログラムの性能計測のためのインタフェースである．対象 JVM に監視プログラムを付加して実行することができ，CPU の時間消費やメモリの使用量を調べることができる．メソッドの呼び出しやスレッド，メモリの管理など，主要なプログラムの動作を監視する機能が提供されているが，イベント生成のオーバヘッドが大きく，実行時のコストが高くなるという問題がある．また，JVMPI を用いた監視プログラムは非同期で生成されるイベントを処理するための同期処理を実装する必要がある．さらに，内部でエラーが発生すると監視対象の JVM も同時に異常終了してしまうことから，監視プログラム自体のデバッグが難しいという問題もある．

(c) は，Java を用いてデバッガを作成するための機構とライブラリである．JDI を用いたプログラムは，デバッグ対象のプログラムを実行している JVM の Java Virtual Machine Debugger Interface（JVMDI）と通信し，ブレークポイントの設置，フィールドやメソッド呼び出しイベントの取得，各時点でのスタックフレームの取得など，デバッグのための種々の機能が利用できる．しかし，デバッガはソケットを介した通信を行うほか，JVM の状態を取得するためにプログラム本体の実行を頻繁にブロックすることから，オーバヘッドは大きい．JVMDI を直接扱うこともできるが，その場合でも JVMPI と同様の問題を持つことになる．

(d) は，JVM の公開されたソースコードに手を加えて，プログラムの動作を監視する方法である．この方法は，Java の実行環境におけるすべての情報にアクセスできるという利点がある．しかし，JVM の実装に依存し，JVM のバージョンアップへの対応が必要である．また，(b)，(c)，(d) 共通して，バイトコードレベルでの処理が必要であり，Just In Time（JIT）コンパイラによる最適化を行うと，得られる結果が変わってしまう可能性がある．そのため，最適化の抑止が必要となり，結果としてパフォーマンス上のオーバヘッドが生じる．

これらに対し，アスペクトによるプログラム解析の実現は，抽象的な Join Point という形式でプログラムの結合を行うことができるため，(a) の持つ問題点の影響を受けない．また，アスペクトは実行環境ではなくプログラムを変換するため，(b)，(c)，(d) が持つ JVM への依存性の影響を受けずに済むという利点がある．また，アスペクトが結合されたプログラムは標準の Java プログラムとなるため，小さなプログラムに対してアスペクトを結合し，Java 用のデバッガなどを用いてアスペクトの持つ欠陥を除去することが容易である．

3.4 AspectJ による動的解析の実現

AspectJ の持つ，アスペクトが結合されているソースコードの位置情報へアクセスする機能を用いることで，フィールドの参照や代入，メソッド呼び出しに対して，プログラム内の依存関係の解決を行うことができる．

AspectJ を用いると，データ依存解析および動的束縛解決のアルゴリズムは次のように記述することができる．

- **データ依存関係の解決**

  フィールドへの値の代入 代入されたオブジェクトへの参照と，そのフィールドのシグネチャ，代入文の位置を記録する．

☆ 著者らの知る限り，JDI を用いたプログラム実行時情報解析の手法は提案されていない．

```
public aspect DataDependsAnalysisAspect {

  pointcut target():
    !within(slice.aspect.*);

  pointcut exclude():
    within(somepackage.*);

  pointcut field_set():
    target() && !exclude() &&
    (set(* *) || set(static * *));

  pointcut field_get():
    target() && !exclude() &&
    (get(* *) || get(static * *));

  FieldDef def = new FieldDef();

  before(): field_set() {
    def.put(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      thisJoinPoint.getSourceLocation());
  }
  before(): field_get() {
    SourceLocation setpos =
      def.get(thisJoinPoint.getTarget(),
              thisJoinPoint.getSignature());
    Logger.logDataDepends(
      thisJoinPoint.getTarget(),
      thisJoinPoint.getSignature(),
      setpos,
      thisJoinPoint.getSourceLocation());
  }
}
```

図 4 データ依存関係解析の実装（抜粋）
Fig. 4 A piece of the implementation of dynamic data dependence analysis.

フィールドの値の参照 参照されたオブジェクトと，そのフィールドのシグネチャに一致する代入文の位置を取得し，参照した文の位置へのデータ依存関係を記録する．

- **動的束縛の解決**

メソッド呼び出し スレッドごとに用意されたスタックへ，メソッド呼び出し位置と呼び出したメソッドの内容を記録する．

メソッド実行 スレッドごとに用意されたスタックを見て，呼び出し位置から，実際に呼び出されたメソッドへの制御依存関係を記録する．

メソッド呼び出し終了 スレッドごとに用意されたスタックから，呼び出し情報を取り除く．

例外の発生 メソッド呼び出し終了と同様の処理を行う．

データ依存関係の解決を行うコードの抜粋を**図 4** に示す．動的束縛の解決については，図 3 に示した例をマルチスレッドに対応させたものとなっているため，ここでは省略する．

依存関係解析アスペクトは，AspectJ のワイルドカード指定機能を用いて，解析対象となるクラスのすべてのフィールド参照と代入に対して動作するように定義している．この実装は，利用者がアスペクトのコードを操作せずに利用するためのものである．しかし，ユーザが監視対象から外したいクラスがある場合は，AspectJ の継承機能を用いて，監視対象から除外したいクラスを再定義した新しいアスペクトを作成することができる．

アスペクトが対象プログラムの本来の振舞いを破壊することはない．アスペクトの結合によって，対象となるプログラムのデータフローと制御フローが変化する．しかし，データフローについては，対象プログラムの値を観測および記録はするが値を書き換えることはない．また，オブジェクトを弱参照で取り扱うため，オブジェクトの生存期間にも影響を与えることはなく，対象プログラムの振舞いを変更することはない．弱参照とは，オブジェクトへの他のすべての参照がすべて破棄されると自動的に破棄されるような参照の機構であり，Java では言語の標準機能として提供されている．一方，制御フローに関しては，単純な実装では無限ループが発生する可能性があるため，それを回避するための実装を行うことで，プログラムの本来の振舞いへの影響を回避している．この問題については，実装上の制限として 3.5.3 項で詳しく説明する．

3.5 AspectJ における実装上の制限

3.5.1 Join Point の制限

アスペクト指向プログラミングでは，利用可能な Join Point と，それに対して適用可能な演算によってアスペクトの記述可能な範囲が制限される．AspectJ では，ローカル変数の読み書きや制御構造は Join Point として含まれない．これは，ローカル変数や制御構造に対する横断処理が必要とされるケースが少ないこと，またパフォーマンス上著しいオーバヘッドを引き起こすことに起因する．

動的データ依存解析では，本来ならばすべての変数における値の授受を監視しなければならないため，AspectJ では厳密な実装は不可能である．しかし，ローカル変数に関するデータ依存関係は単一の手続き内で完結しているため，オブジェクト指向における実行時決定要素の影響を受けにくく，静的に解析しても十分な精度を得られると予測される．この件に関しては，後述する適用実験の考察で議論する．

3.5.2 ソースコードの制限

AspectJ はソースコードに対してアスペクトの結合を行うため，ライブラリに対してはアスペクトの結合

を行うことはできない．ここでライブラリとは，Java ソースコードが存在しないバイナリ形式の再利用可能なコンポーネントを指す．

これに対して，本研究では，以下の理由からライブラリは解析対象から除外する方針を採った．

ライブラリの信頼性は高い． ライブラリは再利用の単位であり，その内部は十分に信用できるコードであると考えられる．そのため，必要以上に詳細な解析は必要ない．

ライブラリのコード量は非常に多い． ライブラリの量は，利用するプログラムと比較して非常に多く，動的に解析するコストが高くなる．

プログラムがライブラリ側からのコールバックを利用する場合，プログラムのある地点からライブラリ内部を経由してプログラムの別の地点へと，隠れた依存関係を生じることがある．これは Java バイトコードでの依存関係解析を行うことで知ることができる[3]．しかし，ファイル入出力やデータ構造のような基本的なオブジェクトに対しては，後述する Java 言語上の制限から，バイトコードを用いても依存関係解析を行うことはできない．この問題に対しては，ソースコードが存在する範囲での動的解析と，解析できない範囲への静的解析の組合せで対処する．

ライブラリに対するメソッド呼び出し文については，呼び出し側の情報は動的に取得できるが，実際に呼ばれたメソッドの中での動作は解析できない．そこで，メソッドが呼び出されたオブジェクトとメソッドの引数から，そのメソッドの戻り値へのデータ依存関係があると見なして対処する．また，呼び出した結果，そのクラスを経由して他の観測対象となるメソッドが 1 つ以上呼び出される可能性もある．これについては，ライブラリへの呼び出し文から，実際に呼ばれたメソッド群に対しての呼び出し関係を記録し，依存関係を設定する．

3.5.3 Java 言語上の制限

AspectJ ではアスペクトを Java で平易に記述できるという利点があるが，アスペクトにも，データの収集に利用するクラスに対して依存関係が生じてしまう．そのため，モジュールが利用しているクラスに対してアスペクトを結合して解析しようとすると，ループが生じることがある．

ループの発生例を**図 5** に示す．この図では，メソッド Foo.getX を呼び出すが，そのメソッド呼び出しに対応してアスペクトが作動する．アスペクトは Foo に対してハッシュコードを要求するが，Foo.hashCode が getX メソッドを用いて計算されている場合，getX

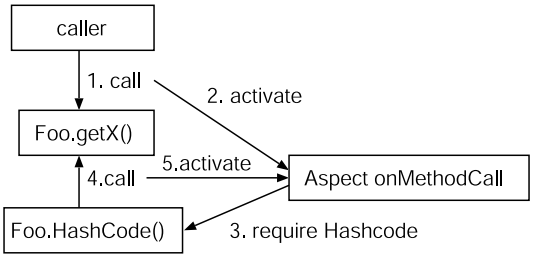

図 5 ループ発生例
Fig. 5 Loop occurence by aspect.

呼び出しにより再びアスペクトが作動してループに陥ってしまう．

このループの発生の問題は，バイトコードを加工するアプローチであっても同様で，JVM 改造アプローチのような言語の枠を越えた手段を用いない限り本質的に解決することはできない．

しかし，Java 標準ライブラリのクラスに対する解析を行わない限り，ループの原因となるメソッドは限られる．具体的には，アスペクトから標準ライブラリを通じて間接的に呼ばれるオブジェクトの文字列表現への変換（Object.toString），ハッシュコードの計算（Object.hashCode）の 2 つである．アスペクトから toString，hashCode への呼び出しを避けること，また toString，hashCode へのアスペクトの結合を避けることでこの問題を回避することができる．この対処は toString，hashCode について収集する情報を制限してしまうが，これらのメソッドの役割は，通常そのメソッドだけで完結しているので，このような原因による情報の完全性の低下は，実用上の影響を与えないと考えられる．

4. 評価実験

4.1 概　　要

AspectJ で作成した動的依存解析モジュールを使って，Java を対象とした DC スライスを計算するシステムを構築した．**図 6** にシステムの概略を示す．

ユーザは，AspectJ コンパイラを用いて解析対象の Java のソースコードを動的依存解析アスペクトと結合する．生成されたクラスファイルは Java 標準のバイトコードなので，通常の JVM で実行することができる．動的依存解析アスペクトは，プログラムが終了する際に解析結果をファイルに書き出す．この解析結果を，ソースコードとあわせて与えることで，プログラム依存グラフを作成することができ，ユーザが任意の DC スライスを計算することができる．

このシステムを用いて，計算される DC スライス

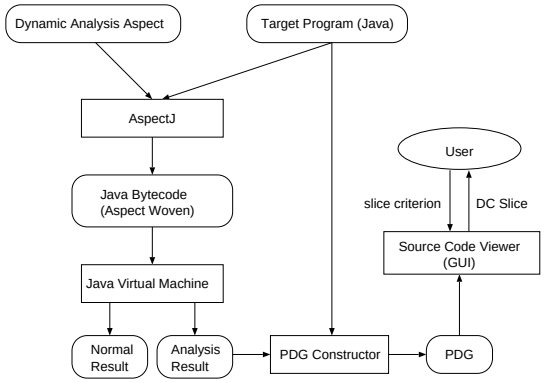

図 6 システム概略
Fig. 6 System overview.

表 3 適用対象
Table 3 Target programs.

	種別	クラス数	サイズ（LOC）
P1	簡易データベース	4	262
P2	ソーティング	5	228
P3	DC スライス計算	125	16207

のサイズと動的依存関係解析に必要な時間コスト，モジュールサイズについて，JVM 改造アプローチ[3] との比較実験を行った．ここで使用した改造 JVM は，DC スライスを計算するために，すべてのデータに関するデータ依存関係の解析を行っている．バイトコード上で依存関係解析を行うため，ソースコードを持たないライブラリに対しても解析を行っていることが特徴である．

適用対象のプログラムを，**表 3** に示す．P1 は簡易データベースプログラムで，オブジェクト指向言語にある特有な要素をほとんど利用していない．P2 はソーティングプログラムで，配列，オブジェクトの多態性などを用いている．P3 は今回開発した DC スライス計算プログラムで，ライブラリに対する多数のメソッド呼び出し，クラスの多態性と動的束縛，例外処理，対話的なユーザインタフェースなど，Java の特徴的な要素を数多く備えている．

これらのプログラムに対していくつかの入力を与えて実行し，DC スライス計算を行った．

以降，適用結果を基に，4.2 節でスライスサイズについて，4.3 節で時間コストについて，4.4 節で計測プログラムのモジュールサイズについて，考察を述べる．

4.2 スライスサイズの比較

表 4 に，P1，P2，P3 からそれぞれ選んだスライス基点 S1，S2，S3 における DC スライスのサイズ（LOC）を示す．どのプログラムもファイルあるいは

表 4 スライスサイズ [LOC]
Table 4 Slice size.

スライス基点	改造 JVM	Aspect	Aspect/JVM
S1 (P1)	29	36	1.24
S2 (P2)	28	50	1.79
S3 (P3)	708	839	1.19

GUI に対するデータ出力を行うため，出力されるデータに対応する変数の 1 つを基点として選択した．

プログラムスライスの計算では，一般的に，プログラムスライスに含まれるべき文は少なくとも含まれるように計算する．スライスサイズの差は，正確性の差を示すことになる．

アスペクトによるアプローチは，ローカル変数に関しては静的解析で補うことなどが原因で,「依存する可能性がある」文がスライスに加わることになる．プログラム中に含まれる条件節のうち，条件が成立しないために実行されないような文がある場合，JVM 改造アプローチでは実行されていない文を除去するが，アスペクトによる実現では，ローカル変数の依存関係から文をスライスに含める場合がある．

S1 では，プログラムサイズが小さいため，実質的な差は現れなかった．一方，S2 では大きな差が発生した．スライスの内容を確認したところ，長いメソッドが多く，ローカル変数の依存関係が多いことが原因となっていた．S3 はプログラムが適切なサイズのモジュールに分解されており，ローカル変数が多数使われたメソッドは少なく，S1 と同程度の増加となった．

一度も実行されなかった文が，静的な解析によってスライス結果に含まれてしまった場合でも，そこに含まれたメソッド呼び出しやフィールド参照については実行時情報が存在しないため，他のメソッドなどへ依存関係の追跡が波及していくことはない．結果として，そのブロックの範囲にスライスの増加は限られる．この差異が実際の開発作業に与える影響の評価については，今後の課題として検討していく．

また，制御フロー情報を用いて，実行されなかったことが判明している文の前後の文をスライスから取り除くことで，精度を向上できる可能性があるが，これについても今後の課題とする．

4.3 解析コストの比較

通常の場合，改造 JVM の場合，アスペクトを結合した場合とで，プログラムに同一の入力を与え，実行した場合の動作にかかった時間を，JIT コンパイラによる最適化なしで比較したものを**表 5** に示す．また，JIT コンパイラを有効にした場合での，通常のプログラムの実行時間とアスペクトを結合したプログラムの

表 5 実行時間 [秒]
Table 5 Execution time [sec.].

適用対象	通常	改造 JVM	アスペクト
P1	0.18	1.8	0.26
P2	0.19	2.8	0.39
P3	1.2	81.0	10.3

表 6 JIT を有効にした場合の実行時間 [秒]
Table 6 Execute time, JIT enabled [sec.].

適用対象	通常	アスペクト
P1	0.24	0.34
P2	0.24	0.41
P3	1.1	9.9

実行時間との比較を**表 6** に示す．

一般に，アスペクトによる実装のほうが改造 JVM 側に比べて高いパフォーマンスを発揮した．P1 と P2 ではライブラリをほとんど用いていないため，ローカル変数の動的解析のコストが高いことがその差の影響であると考えられる．また，P3 では，Java のソースコードを構文解析するために用いているライブラリの内部処理に対しても解析を行っていることが，さらなるコスト増加の要因となっている．大規模なプログラムになるほど，ライブラリは多く用いられるため，コスト増加の傾向はさらに強くなると考えられる．

アスペクトを用いた際の利点である JIT コンパイラによる最適化の影響は，小規模なプログラムでは最適化に要するコストのほうが高くつくため，P1 や P2 では実行時間の増大を招いた．しかし，P3 のようにある程度の規模を持つプログラムでは，最適化の恩恵を受けることができる．この影響はプログラムや実行環境に依存するため一概にはいえないが，パフォーマンス上重大な差異を与える可能性が，実験的に示されている[10]．

4.4 スライスツール実装の比較

アスペクトとして記述したデータ依存解析モジュールは，400 行程度であった．また，DC スライス計算ツールは Java を用いて約 16,000 行で記述することができた．

アスペクトによるアプローチでは，プリプロセッサに比較して高い抽象度で可読性の高い記述が可能であるほか，モジュールのサイズが小さいために，後から解析するために必要な情報を記録するだけにとどめる，あるいは実行時にすべての解析を行って不要なデータを捨てていく，といったように実行環境にあわせて実装を柔軟に切り替えることが容易である．

JVM 改造によるアプローチでは，Java のコンパイラと JVM，あわせて 50 万行以上のプログラムに対して約 16,000 行のコードを追加している[4]．この追加コードは，まず，ローカル変数も含むすべてのデータ依存関係を取得するという点でアスペクトによるアプローチよりも実装量が多い．それに加えて，Java バイトコードからソースコードへのマッピング情報を計算するためのコードが含まれている．バイトコードからソースコードへのマッピングは，スライス結果を表示する際に使用される．

このようなソースコードのサイズの違いだけでなく，JVM 改造アプローチでは JVM のバージョンアップや利用可能な実行環境にあわせて調整していく必要が生じるため，実現に要するコストは多大なものとなる．これに対して，アスペクトによるアプローチでは言語仕様に変更が加わらない限り，自由な環境で利用していくことができるため，実現コストを大幅に低減することができる．

5. ま と め

アスペクト指向プログラミングを用いて，プログラムスライスに必要な動的データ依存解析処理を実現した．

アスペクト指向プログラミングは，オブジェクトを横断した要素をアスペクトという新しいモジュール単位として独立記述することを可能とする．アスペクトをオブジェクト指向プログラムに結合するには，Join Point と呼ばれる結合基準を用いる．これは通常のプリプロセッサで用いられる抽象構文木による表現と比較して，より抽象度が高い形式での記述を可能とする．また，マルチスレッドや例外処理といった，高度な言語機能に対する処理を記述することも容易となる．

アスペクトの結合基準を一般的に記述することで，動的データ依存解析アスペクトは，様々なオブジェクト指向プログラムに対して，モジュールを修正することなく結合を行うことができる．これによって，従来の JVM 改造アプローチなどに比べ，動的依存解析処理の保守性，再利用性が向上した．

今回，システムの実現には AspectJ を用いたが，AspectJ ではローカル変数に対する動的解析を記述できないという制約があった．JVM 改造アプローチとの比較実験によって，この制約によってスライスサイズが増大するが，実行時間のオーバヘッドを著しく削減し，高い保守性を実現できることを示した．

また，アスペクトによる実現では，Java の実行環境や言語仕様に対する依存性は抑えられており，他のプログラミング言語に対しても，適切な Aspect Weaver を利用してアスペクトを記述することで動的依存解析

を実現することができる．

今後の課題としては，アスペクトによる解析でのスライスサイズの増大が実際の開発作業に与える影響の評価，静的解析では依存関係があるが実際には実行されていない（実行時情報が存在しない）メソッド呼び出し文などを手がかりにした誤差修正，今回の実験では適用していない数十万行程度の大規模プログラムに対してこれらの手法を適用した場合のスケーラビリティについて調査することがあげられる．

参 考 文 献

1) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming, *Proc. 11th European Conference on Object-Oriented Programming*, LNCS, Vol.1241, pp.220–242 (1997).
2) Ashida, Y., Ohata, F. and Inoue, K.: Slicing Methods Using Static and Dynamic Information, *Proc. 6th Asia Pacific Software Engineering Conference*, Takamatsu, Japan, pp.344–350 (Dec. 1999).
3) 誉田謙二，大畑文明，井上克郎：Javaバイトコードにおけるデータ依存解析手法の提案と実現，コンピュータソフトウェア，Vol.18, No.3, pp.40–44 (2001).
4) 誉田謙二：バイトコード間の動的依存情報を抽出するJavaバーチャルマシン，大阪大学大学院基礎工学研究科修士学位論文 (2002).
5) 高田智規，井上克郎，大畑文明，芦田佳行：制限された動的情報を用いたプログラムスライシング手法の提案，電子情報通信学会論文誌 D-I（採録決定）．
6) Agrawal, H. and Horgan, J.: Dynamic Program Slicing, *SIGPLAN Notices*, Vol.25, No.6, pp.246–256 (1990).
7) Ottenstein, K.J. and Ottenstein, L.M.: The program dependence graph in a software development environment, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, pp.177–184 (Apr. 1984).
8) Ueda, R., Inoue, K. and Iida, H.: A Practical Slice Algorithm for Recursive Programs, *Proc.International Symposium on Software Engineering for the Next Generation*, Nagoya, Japan, pp.96–106 (Feb. 1996).
9) Yacoub, S., Ammar, H. and Robinson, T.: Dynamic Metrics for Object Oriented Designs, *Proc. 6th International Symposium on Software Metrics* (*METRICS99*), Boca Raton, Florida, USA, pp.50–61 (1999).
10) Performance Comparison of JIT. http://www.shudo.net/jit/perf/index.html
11) Ohata, F., Hirose, K., Fujii, M. and Inoue, K.: A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information, *Proc. 8th Asia-Pacific Software Engineering Conference*, pp.273–280 (2001).
12) Kusumoto, S., Imagawa, M., Inoue, K., Morimoto, S., Matsusita, K. and Tsuda, M.: Function point measurement from Java programs, *Proc. 24th International Conference on Software Engineering*, pp.576–582 (2002).
13) AspectJ Team: The AspectJ Programming Guide. http://aspectj.org/doc/dist/progguide/
14) Java Platform Debugger Architecture. http://java.sun.com/j2se/1.4/docs/guide/jpda/architecture.html
15) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley (1995).
16) Weiser, M.: Program slicing, *IEEE Trans. Softw. Eng.*, SE-10(4), pp.352–357 (1984).

(平成15年3月3日受付)
(平成15年5月6日採録)

石尾　隆（学生会員）

平成15年大阪大学大学院博士前期課程修了．現在同大学院博士後期課程在学中．アスペクト指向プログラミングおよびプログラム構造解析の研究に従事．

楠本　真二（正会員）

昭和63年大阪大学基礎工学部情報工学科卒業．平成3年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成8年同大学講師．平成11年同大学助教授．平成14年大阪大学基礎工学部情報工学科コンピュータサイエンス専攻助教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IEEE会員．

井上 克郎（正会員）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59 年～61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．工学博士．平成 14 年大阪大学基礎工学部情報工学科コンピュータサイエンス専攻教授．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，IEEE，ACM 各会員．