特別研究報告

題目

プログラム理解における Thin slice の統計的調査による有用性評価

指導教員
井上 克郎 教授

報告者
秦野 智臣

平成 25 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 24 年度 特別研究報告

プログラム理解における Thin slice の統計的調査による有用性評価

秦野 智臣

内容梗概

ソフトウェアの保守作業において，開発者は，多くの時間をプログラム理解に費やしていると言われている．また，その中でも多くの時間を占めるのが静的なデータ依存関係やメソッドの呼び出し関係を調査する作業であると言われている．

プログラム理解の時間を減らすための技術として，プログラムスライシングが古くより提案されている．プログラムスライシングは，プログラム内のある文を基準として，その文に影響を与える可能性のあるすべての文を抽出する技術である．開発者は，抽出された文（プログラムスライス）のみを読むことによって，プログラム理解にかかる時間を減らすことができる．しかし，大規模プログラムの場合は，プログラムスライス自体が非常に大きくなってしまい，プログラム理解において，プログラムスライシングが有用であるとは言い難い．

この問題を解決するプログラムスライシング技術の 1 つとして，Thin slicing が提案されている．Thin slicing は，開発者が選んだ文に影響を与える可能性のあるすべての文ではなく，その文が使用するデータを生成した文のみを抽出するスライシングである．実際に Thin slicing を利用した場合に，従来のスライシングを利用する場合より，プログラム理解にかかる時間が大きく減少した例が紹介されている．しかし，Thin slice の平均サイズを計算する研究は行われておらず，一般に Thin slicing が，どの程度プログラム理解において有用であるのかということは示されていない．

そこで，本研究では，Thin slice の有用性を評価するために，7 個の Java プログラムのすべてのデータフローを対象に Thin slice を計算し，そのサイズに関する統計的調査を行った．その結果，Thin slice のサイズの平均値は十分小さくなることが確認できた．また，プログラム中のデータ使用場所と生成場所を対象にしたスライスの約 75 %が，複数のメソッドにまたがるデータフローを表現することから，多くの場合で，Thin slicing により静的なデータ依存関係の調査の簡略化が期待できることが分かった．

主な用語

プログラム理解
プログラムスライシング
Thin slicing

目 次

1　まえがき

開発者はプログラムの保守作業に多くの時間を費やしており，保守作業の中でも，多くの時間をプログラム理解に費やしていると言われている [5, 6, 13]．さらに，プログラム理解を行うためには静的なデータ依存関係やメソッドの呼び出し関係を調査する作業が必要であり，開発者は，この作業にしばしば時間をかけていることが指摘されている [12, 19]．

この作業の時間を減らすための技術として，プログラムスライシングがある [20]．プログラムスライシングは，プログラム内のある文を基準として，その文に影響を与える可能性のあるすべての文を抽出する技術である．プログラムスライシングによって，開発者がプログラム理解のために読むコードが少なくなり，理解にかかる時間を減らすことができる．

しかし，プログラムスライス（以下，単にスライスという）のサイズの平均値は，プログラム全体の約 30 %であり [3]，大規模プログラムの保守作業におけるプログラム理解の際に，プログラムスライシングを利用することは困難である．

この問題を解決するためのスライシング手法として，Thin slicing が提案されている [16]．Thin slicing は，プログラム内の基準となる文に影響を与える可能性のあるすべての文ではなく，基準となる文が使用するデータを生成した文のみを抽出するスライシング手法である．このため，Thin slice のサイズは大幅に減少する．Thin slicing はその定義から，プログラム理解に必要な作業の中でも，複数のメソッドを経由するデータの依存関係を調査する作業にかかる時間を減らす効果が期待できる．実際に，Thin slicing によりプログラム理解の時間が減少した例が 22 個紹介されている [16]．

従来のスライシング手法については，Binkley らによるスライスサイズの平均値に関する研究 [3] が行われている．また，Jász ら [10] は，メソッド呼び出しと制御依存関係のみを用いたスライスの高速な近似計算を提案しているが，その計算結果の平均値はスライスサイズの平均値より大きくなることが示されている．Thin slicing は，メソッド呼び出しについて従来のスライシングより単純であるが高速な手法を用いており，Binkley らや Jász らの研究のように，Thin slice のサイズの平均値を求めることによって Thin slice の有用性を評価する必要がある．

本研究では，Java 言語で記述されたプログラムを対象とした Thin slicing を実装し，プログラム理解における Thin slice の有用性を統計的調査によって評価する．Thin slicing は，開発者が選択した文で使用するデータを生成した文のみを抽出するスライシングであるので，メソッド呼び出しをたどってデータの生成元を特定する作業を簡略化することが期待できる．また，スライスサイズが十分に小さければ，開発者が調査しなければならない文が少なくなり，プログラム理解にかかる時間を減らすことが期待できる．しかし，メソッド呼び出しをたどらなくともデータを生成している文が容易に見つかる場合は，Thin slicing を用

いる効果は低いと考えられる．そのため本研究の実験では，プログラムにおける全データフローから Thin slicing の効果が期待できるデータフローの割合を調査するために，スライスサイズに加え，スライスがまたがるメソッド数や，使用しているデータの生成元となっている文の数といった指標を計測した．

以降，2 章では研究背景について述べる．3 章では Thin slicing の実装について説明する．4 章では評価実験について述べる．5 章では関連研究について述べ，6 章ではまとめと今後の課題に関して述べる．

2　背景

2.1　プログラムスライシング

プログラムスライシングは，プログラム内のある文に影響を与える可能性のあるすべての文を抽出する技術である [20]．プログラムスライシングには，静的プログラムスライシングと動的プログラムスライシングがある．静的プログラムスライシングは，プログラム P にどのような入力を与えてもプログラム P と同じ動作をするプログラム P の部分集合を出力する．一方，動的プログラムスライシングは，プログラム P' にある入力を与えた場合に，プログラム P' と同じ動作をするプログラム P' の部分集合を出力する．本研究で扱うプログラムスライシングとは，静的プログラムスライシングのことである．プログラムスライシングを実行するために必要なプログラム依存グラフとシステム依存グラフ，またこれらを利用したスライスの計算方法について説明する．

2.1.1　プログラム依存グラフ

プログラム依存グラフ（Program Dependence Graph，以下 PDG）は，プログラム内の 1 つの手続きの代入文と制御文の条件式を頂点とし，それらの文の間の依存関係を有向辺で表したグラフである．依存関係はデータ依存関係と制御依存関係があり， それぞれ次のように定義される．

データ依存関係

文 s_1 と s_2 の間に以下の条件がすべて成り立ったとき，s_2 は s_1 にデータ依存するといい，s_1 の頂点から s_2 の頂点にデータ依存辺を引く．

1. s_1 が変数 x の値を定義する．
2. s_2 が変数 x の値を使用する．
3. s_1 から s_2 に，x の値を書き換えない実行経路が少なくとも 1 つ存在する．

制御依存関係

文 s_1 が制御文であり，s_1 の結果によって s_2 が実行されるかどうかが決定される場合，s_2 は s_1 に制御依存するといい，s_1 の頂点から s_2 の頂点に制御依存辺を引く．

また，手続きの入口を表すエントリ頂点という特別な頂点を設け，その頂点から手続き内の文にも制御依存辺を引く．ただし，すでに手続き内の文から制御依存辺が引かれている頂点には引かない．

PDG の例を図 1 に示す．この図では，代入文の頂点を丸角矩形，制御文の条件式とエントリ頂点をひし形の図形で表している．また，データ依存辺を実線の矢印，制御依存辺を点

```
1  void main() {
2    int sum, i;
3    sum = 0;
4    i = 1;
5    while (i <= 10) {
6      sum = sum + i;
7      i = i + 1;
8    }
9  }
```

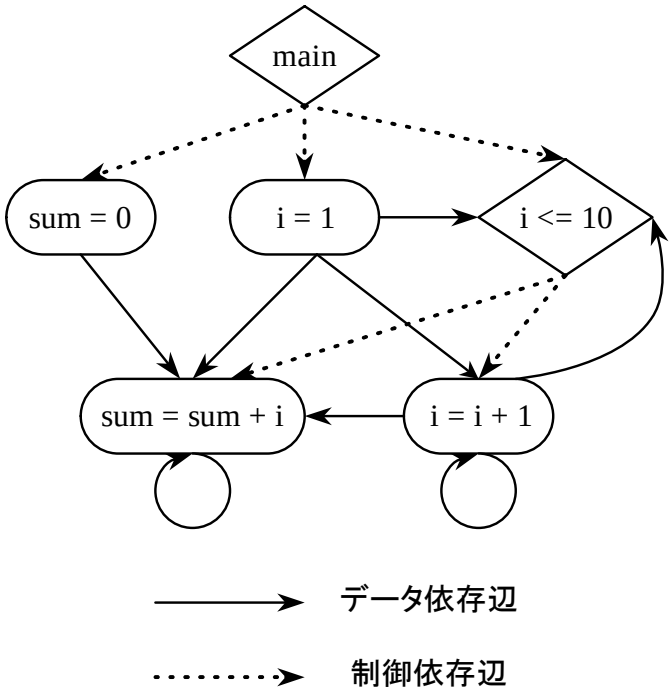

図 1: PDG の例

線の矢印で表している．

2.1.2 システム依存グラフ

システム依存グラフ [8]（System Dependence Graph，以下 SDG）は，プログラム内の各手続きに対応した PDG を手続き間の依存関係で接続したグラフである．手続きの実パラメータと仮パラメータ，あるいは返り値と呼び出し元の依存関係を表すため，特殊変数とその変数の代入文を表す特殊頂点をそれぞれ用意する．手続き間の依存関係は，次の 5 つの頂点とそれらの頂点に関する依存辺によって表される．

call 頂点 手続き呼び出し文を表す頂点．

actual 頂点 手続きの実パラメータから特殊変数への代入文を表す頂点．

formal 頂点 特殊変数から手続きの仮パラメータへの代入文を表す頂点．

return 頂点 手続きの返り値から特殊変数への代入文を表す頂点．

result 頂点 特殊変数から手続きの呼び出し元への代入文を表す頂点．

以上の頂点について，手続き間の依存関係を表す次の依存辺が引かれる．

call 辺 call 頂点から，呼び出される手続きのエントリ頂点への制御依存辺．

parameter 辺 actual 頂点から対応する formal 頂点へのデータ依存辺．

return 辺 return 頂点から result 頂点へのデータ依存辺．

summary 辺 actual 頂点から推移的にデータ依存する result 頂点へのデータ依存辺．

call 頂点，actual 頂点，result 頂点は手続きを呼び出す側に対応する PDG に含まれるものとし，call 頂点から actual 頂点と result 頂点に制御依存辺を引く．また，actual 頂点がデータ依存する同じ手続き内の頂点から，その actual 頂点にデータ依存辺を引き，result 頂点から，その result 頂点にデータ依存する同じ手続き内の頂点にデータ依存辺を引く．formal 頂点と return 頂点は呼び出される手続きに対応する PDG に含まれるものとし，その呼び出される手続きのエントリ頂点から formal 頂点と return 頂点に制御依存辺を引く．また，formal 頂点から，その formal 頂点にデータ依存する同じ手続き内の頂点にデータ依存辺を引く．

SDG の例を図 2 に示す．この図では，パラメータ用の特殊変数を p(x)（x は仮パラメータ名）とし，返り値用の特殊変数を ret としている．それらの特殊変数が含まれる actual 頂点，formal 頂点，return 頂点，result 頂点を長方形の図形で表している．また，各頂点が所属する手続きを点線の枠で表している．

call 頂点，actual 頂点，result 頂点は手続き呼び出し文の数だけ設けるが，各手続きの PDG はそれぞれ 1 つであるため，複数の actual 頂点から 1 つの formal 頂点に依存辺が引かれることがある．Java の場合，フィールド変数に関する頂点を引数頂点として表現することが提案されている [7]．

2.1.3 スライスの計算

スライスは，スライスを計算するための基点となる文と変数を定め，その基準（これをスライシング基準という）に相当する SDG 上の頂点から依存辺に沿った探索を行い，スライシング基準の頂点自身を含む到達範囲の頂点集合として計算される．スライスの計算は，依存辺をたどる向きによって後ろ向きスライシングと前向きスライシングに分けられており，summary 辺 [15] を使用した Two-Phase Slicing と呼ばれる効果的な探索のルールがそれぞれに定義されている [8]．後ろ向きスライシングの場合は，次のような手順で探索を行い，到達したすべての頂点の集合がスライスとなる．

Phase1 スライシング基準に相当する頂点から return 辺を除くすべての依存辺を後ろ向きにたどる．

```
1  void main() {
2    int sum, i;
3    sum = 0;
4    i = 1;
5    while (i <= 10) {
6      sum = add(sum, i);
7      i = i + 1;
8    }
9    print(i);
10   print(sum);
11 }
12 int add(int x, int y) {
13   return x + y;
14 }
```

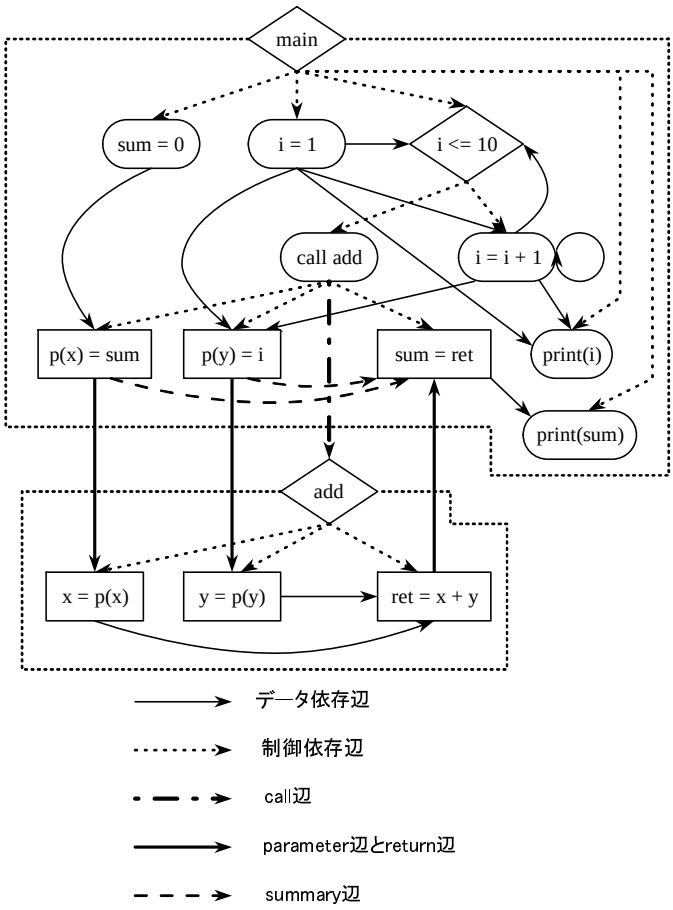

図 2: SDG[8] の例

Phase2 Phase1 で到達した頂点から call 辺と parameter 辺を除くすべての依存辺を後ろ向きたどる．

後ろ向きスライシングは，スライシング基準の実行に影響を与える可能性のあるすべての文の集合を出力する．図 2 の 9 行目の文 print(i) と変数 i をスライシング基準として後ろ向きスライシングを実行すると，スライスは {4，5，7，9} 行目となる．前向きスライシングは，依存辺を順方向にたどることで，スライシング基準の実行に影響を受ける可能性のあるすべての文の集合を出力する．図 2 の 4 行目の文 i = 1 と変数 i をスライシング基準として前向きスライシングを実行すると，スライスは {4，5，6，7，9，10，13} 行目となる．

2.1.4 スライスサイズに関する研究

プログラムスライシングで出力されるスライスについて，Binkley らの研究 [3] がある．この研究では，合計 136,000 行に及ぶ 43 のプログラムについて，すべての文に対して前向きスライスと後ろ向きスライスを計算した実験が行われており，スライスに含まれる文の平均は，プログラム内のすべての文の約 30 %であることが述べられている．

この結果は，100 万行の大規模プログラムにおいて，ある変数の値について調べたい場合にプログラムスライシングを利用しても，平均約 30 万行のコードを読む必要があることを示している．大規模プログラムでは，プログラムスライシングを利用しても，プログラム理解にかかる時間は膨大である．

2.2 Thin slicing

Thin slicing[16] は，スライスサイズを減らし，プログラム理解やデバッグ作業における開発者の負担を軽減するために提案されたスライシング手法である．Thin slicing には，Context-Insensitive Thin slicing と Context-Sensitive Thin slicing の 2 種類の計算方法があるが，本研究では Context-Insensitive Thin slicing のみを扱う．これは，Context-Sensitive Thin slicing の計算時間が膨大であり，実用的なプログラムで用いることが現実的ではないためである．

Tripp らの研究 [18] では，Web アプリケーション上で入力された不正なデータが到達する文を特定するために，前向き Thin slicing を利用している．

2.2.1 Thin slicing の定義

従来のプログラムスライシングは SDG 中のすべての依存辺をたどって頂点を探索するが，Thin slicing は SDG 中のデータ依存辺のみをたどって頂点を探索する．さらに，探索でたどるデータ依存辺を以下のように限定する．

- ポインタ変数を介してヒープ領域の変数にアクセスしている文の頂点に関するデータ依存辺について，ヒープ領域の変数に関するデータ依存辺のみを考える．
- 手続き間のデータ依存辺について，parameter 辺と return 辺のみを考える．

また，フィールド変数については parameter 辺や return 辺と同様に，他の手続きであってもデータ依存辺を引く．

2.2.2 Thin slice の計算

Thin slice は従来のスライスと同様に，スライシング基準に相当する SDG 上の頂点から依存辺に沿った探索を行い，到達範囲の頂点集合として計算するが，探索でたどる依存辺を 2.2.1 項のように限定する．

Thin slicing と従来のスライシングの違いを図 3 の例で示す．この図では新たに，ポインタ変数のデータ依存辺を表す破線が追加されている．また，エントリ頂点とその制御依存辺は省略している．この例について，スライシング基準を文 v = z.f で値が代入される変数 v とし，その後ろ向き Thin slicing と従来の後ろ向きスライシングの出力結果について説明する．

まず，従来のスライシングの場合は，図 3 のすべての辺をたどるので，スライスはプログラム内のすべての文となる．一方，Thin slicing の場合は実線の矢印で示したデータ依存辺のみをたどるので，14 行目の文 v = z.f の頂点を基点とし，文 w.f = c，c = a + b，a = 1，b = 2 の頂点を探索する．そのため，スライシング基準 v = z.f の変数 v に対する Thin slice は {6，7，10，12，14} 行目となる．この例のように Thin slice は，スライシング基準で使用する値を生成した文の集合となる．

2.2.3 プログラム理解における利用

図 3 の例のように，Thin slicing は，従来のスライシングよりスライスサイズが小さくなり，使用している変数の値を生成した文がより明確になる．プログラム理解において開発者がデータ依存関係を調査する際に，この特徴を利用することによって理解にかかる時間を減らすことが期待できる．

特に，ソースコード中のある文で使用しているデータについて開発者が調べたい際に，そのデータが様々なメソッドを経由して到達している場合や多くの文をたどって到達している場合は Thin slicing の効果が高い．例えば，図 4 において，32 行目で出力している変数 id の値について調べたい場合，クラス X の addData メソッドの呼び出し元をたどっていかないと変数 id の値を生成した文は分からないが，Thin slicing を利用すると 7 行目と 9 行目で生成された値を使用していることが特定できる（詳細は 3 章の図 12 を参照）．

```
 1  package slice;
 2  public class Main {
 3    public static void
       main(String[] args) {
 4      A x, z, w;
 5      int a, b, c;
 6      a = 1;
 7      b = 2;
 8      x = new A();
 9      z = x;
10      c = a + b;
11      w = x;
12      w.f = c;
13      if (w == z) {
14        int v = z.f;
15      }
16    }
17  }
18  class A {
19    int f;
20  }
```

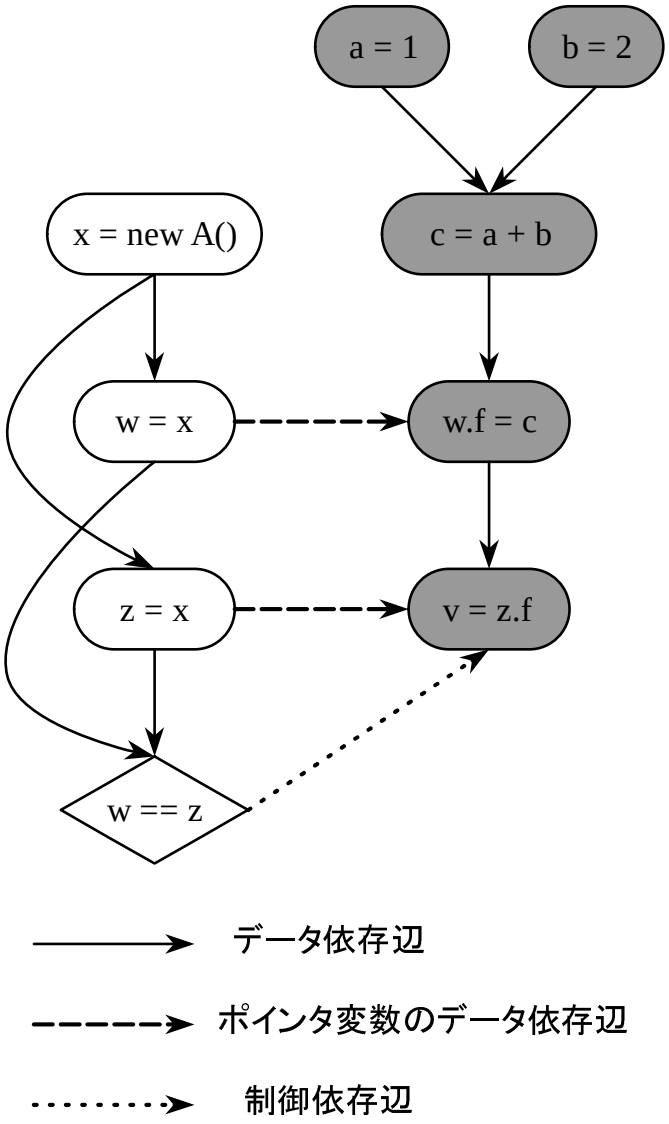

図 3: サンプルプログラム

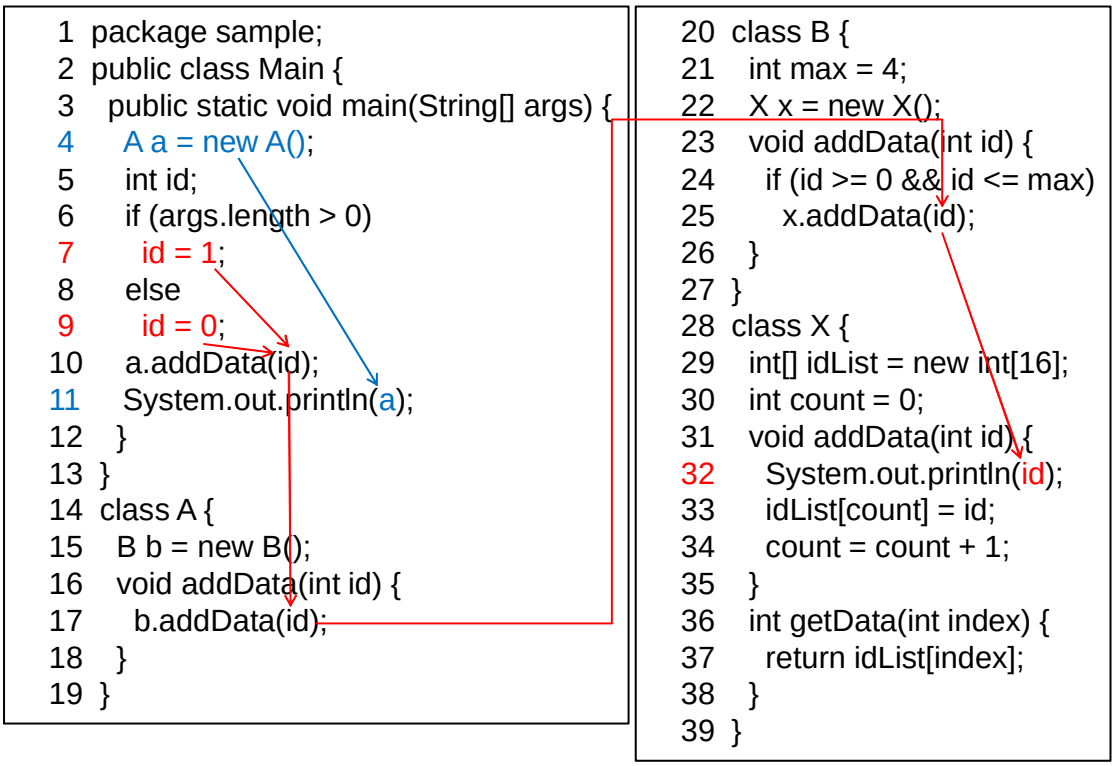

図 4: サンプルプログラム

しかし，調べたいデータについて，そのデータが 1 つのメソッド内で少ない文をたどって到達している場合は Thin slicing の効果が低い．例えば，図 4 において，11 行目で出力している変数 a の値について調べたい場合は，その周辺のコードを読めば Thin slicing を利用しなくとも，4 行目で生成された値を使用していることが特定できる．

```
0: (L1040108132)
1: (line=4)
2: NEW
3: DUP
4: INVOKESPECIAL
     sample/A#<init>()V
5: ASTORE 1 (a)
6: (L580487944)
7: (line=6)
8: ALOAD 0 (args)
9: ARRAYLENGTH
10: IFLE L242666487
11: (L424201356)
12: (line=7)
13: ICONST_1
14: ISTORE 2 (id)
15: (L1586482837)
16: GOTO L860080307
17: (L242666487)
18: (line=9)
19: FRAME-OP(1)
20: ICONST_0
21: ISTORE 2 (id)
22: (L860080307)
23: (line=10)
24: FRAME-OP(1)
25: ALOAD 1 (a)
26: ILOAD 2 (id)
27: INVOKEVIRTUAL
      sample/A#addData(I)V
28: (L657291792)
29: (line=11)
30: GETSTATIC java/lang/System#out
            : java/io/PrintStream
31: ALOAD 1 (a)
32: INVOKEVIRTUAL
      java/io/PrintStream#println
      (Ljava/lang/Object;)V
33: (L447267976)
34: (line=12)
35: RETURN
36: (L2053965899)
```

図 5: 図 4 の main メソッドのバイトコード

3 Thin slicing の実装

本研究では，実験を行うために Java プログラムのバイトコードを対象とした Thin slicing を実装した．バイトコードはソースコードに比べ，プログラムによる解析が容易であるため，本研究の実装ではバイトコードを用いる．本実装では，バイトコードの 1 命令を頂点とした SDG を作成する．本研究で扱うバイトコードの例を図 5 に示す．このバイトコードは，図 4 の main メソッドに対応しており，そのメソッド内でのインデクスと命令を表している．ただし，FRAME-OP や括弧付きで表現されている（L1040108132）と（line=4）などの行は，条件分岐用のラベルとソースコード中の行番号を表すもので，バイトコードではない．以降では，バイトコード上のデータ依存関係について説明する．

3.1 ローカル変数とオペランド・スタックのデータ依存関係

Java 仮想マシンは，オペランド・スタックと呼ばれるスタックに，演算で使用する値やその結果を保持している．そのため，ローカル変数とオペランド・スタック間で値を移動させる命令が存在し，オペランド・スタック上の演算におけるデータ依存関係だけでなく，ローカル変数とオペランド・スタック間のデータ依存関係が存在する．図 6 は 2 つの値を加算するプログラムであり，図 7 はそのバイトコードである．図 7 のインデクス 0，2，8，10 は，オペランド・スタックに定数をプッシュする命令であり，インデクス 1，3，7，9，11，15，19 は，オペランド・スタックからローカル変数に値を移動させる命令である．インデクス 4，5，12，13，16，17 は，ローカル変数からオペランド・スタックに値を移動させる命令であり，インデクス 6，14，18 は，オペランド・スタックから 2 つの値をポップし，加算した結果をオペランド・スタックにプッシュする命令である．この例における各命令間のデータ依存関係は図 8 のようになる．図 8 のグラフにおける頂点の数字はバイトコードのインデクスを表している．

3.2 フィールド変数のデータ依存関係

フィールド変数の依存関係は，次の 2 つの方法で決定する．1 つ目は，同じクラスの同名のフィールドについて，フィールド変数に値を書き込む命令である PUTFIELD とフィールド変数の値を読み込む命令である GETFIELD が行われている場合に，PUTFIELD から GETFIELD にデータ依存辺を引く．2 つ目は，同じクラスの同名のフィールドについて，PUTFIELD と GETFIELD が行われている場合に，そのフィールドが参照しているオブジェクト変数のエイリアスを調べ，エイリアスされている可能性があれば，PUTFIELD から GETFIELD にデータ依存辺を引く．このエイリアス解析は，Andersen のポインタ解析 [2] に基づいた Object-sensitive Pointer Analysis[14] を用いた．図 9 のクラス X のフィールド変数 count について，クラス X の addData メソッドのインデクス 20 で PUTFIELD を行い，addData メソッドのインデクス 10，17 で GETFIELD を行っている．そのため，インデクス 20 の頂点からインデクス 10，17 の頂点にデータ依存辺が引かれる．

```
int x = 1;
int y = 2;
int z = x + y;
int a = 3;
int b = 4;
int c = a + b;
int w = z + c;
```

図 6: 2 つの int 値を加算するプログラム

```
0: ICONST_1
1: ISTORE 1 (x)
2: ICONST_2
3: ISTORE 2 (y)
4: ILOAD 1 (x)
5: ILOAD 2 (y)
6: IADD
7: ISTORE 3 (z)
8: ICONST_3
9: ISTORE 4 (a)
10: ICONST_4
11: ISTORE 5 (b)
12: ILOAD 4 (a)
13: ILOAD 5 (b)
14: IADD
15: ISTORE 6 (c)
16: ILOAD 3 (z)
17: ILOAD 6 (c)
18: IADD
19: ISTORE 7 (v)
```

図 7: 図 6 に対応するバイトコード

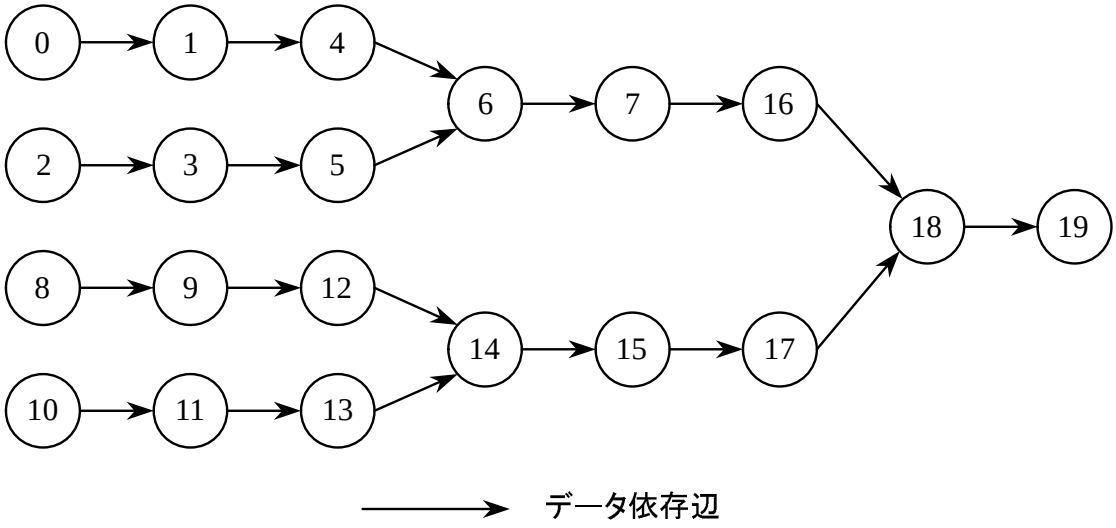

図 8: 図 7 のデータ依存関係

3.3 配列変数のデータ依存関係

配列変数に値を書き込む命令（表 1）から値を読み込む命令（表 2）にデータ依存辺を引く．図 9 の例では，クラス X の addData メソッドのインデクス 12 で IASTORE を行い，クラス X の getData メソッドのインデクス 5 で IALOAD を行っている．このとき，addData メソッドのインデクス 12 の頂点から getData メソッドのインデクス 5 の頂点にデータ依存辺が引かれる．

3.4 メソッドの動的束縛の解決

メソッド呼び出し文において，呼び出されるメソッドが動的に決まる場合は，呼び出される可能性のあるすべてのメソッドに対してメソッド間の依存辺を引く．メソッド間の依存関係については，メソッドを呼び出す側の actual 頂点から，呼び出される可能性のある各メソッドの formal 頂点にデータ依存辺を引く．また，呼び出される可能性のある各メソッドの return 頂点から，呼び出す側の result 頂点にデータ依存辺を引く．このメソッド間の依存関係の決定に必要なメソッドの動的束縛の解決には，Variable-Type Analysis[17] を用いた．

表 1: 配列変数に値を書き込む命令

IASTORE	LASTORE	FASTORE	DASTORE
AASTORE	BASTORE	CASTORE	SASTORE

クラス X の addData メソッド

```
0: (L1065330270)
1: (line=32)
2: GETSTATIC
   java/lang/System#out
   : java/io/PrintStream
3: ILOAD 1 (id)
4: INVOKEVIRTUAL
   java/io/PrintStream#
   println(I)V
5: (L1434682851)
6: (line=33)
7: ALOAD 0 (this)
8: GETFIELD
     sample/X#idList: int[]
9: ALOAD 0 (this)
10: GETFIELD sample/X#count: int
11: ILOAD 1 (id)
12: IASTORE
13: (L768288241)
14: (line=34)
15: ALOAD 0 (this)
16: DUP
17: GETFIELD sample/X#count: int
18: ICONST_1
19: IADD
20: PUTFIELD sample/X#count: int
21: (L1948780723)
22: (line=35)
23: RETURN
24: (L1884511064)
```

クラス X の getData メソッド

```
0: (L1147894048)
1: (line=37)
2: ALOAD 0 (this)
3: GETFIELD
     sample/X#idList: int[]
4: ILOAD 1 (index)
5: IALOAD
6: IRETURN
7: (L1809663735)
```

図 9: クラス X のメソッドのバイトコード

表 2: 配列変数の値を読み込む命令

IALOAD	LALOAD	FALOAD	DALOAD
AALOAD	BALOAD	CALOAD	SALOAD

```
 1  package binding;
 2  public class Bind {
 3    public static void main(String[] args) {
 4      C c;
 5      if (args.length > 0)
 6        c = new C1();
 7      else
 8        c = new C2();
 9      int x = 0;
10      int v = c.getValue(x);
11      System.out.println(v);
12    }
13 }
14  abstract class C {
15    abstract int getValue(int x);
16  }
```

```
17  class C1 extends C {
18    int getValue(int x) {
19      return x + 1;
20    }
21  }
22  class C2 extends C {
23    int getValue(int x) {
24      return x + 2;
25    }
26  }
27  class C3 extends C {
28    int getValue(int x) {
29      return x + 3;
30    }
31  }
```

図 10: メソッドの呼び出し先の候補が複数ある例

図 10 の例では，10 行目で実行される getValue メソッドについて，クラス C1 とクラス C2 両方の getValue メソッドが呼び出されるものとしてメソッド間の依存辺を引く．ただし，クラス C3 の getValue メソッドは呼び出されないことが静的な解析で分かるため，このメソッドには依存辺を引かない．クラス C1 とクラス C2 の getValue メソッドと，main メソッドの 9，10 行目に対応するバイトコード，さらにその SDG を図 11 に示す．

3.5 オペランド・スタック管理命令

オペランド・スタックを直接操作するための命令（表 3）については，本研究の実装では，命令の影響をデータ依存関係の計算に直接反映する．たとえばメソッド呼び出し命令の戻り値が使用されない場合，バイトコード上ではその値を破棄する POP 命令が使用されるが，メソッド呼び出し命令から POP 命令へとデータ依存辺を引く代わりに，メソッド呼び出し命令から戻り値に関するデータ依存辺が存在しないものとして扱う．そのため，表 3 の命令から依存辺が引かれたり，これらの命令に依存辺が引かれることはない．

表 3: オペランド・スタック管理命令

POP	POP2	DUP	DUP2	DUP_X1
DUP2_X1	DUP_X2	DUP2_X2	SWAP	

main メソッドの 9，10 行目

```
21: (line=9)
22: FRAME-OP(1)
23: ICONST_0
24: ISTORE 2 (x)
25: (L1126526626)
26: (line=10)
27: ALOAD 1 (c)
28: ILOAD 2 (x)
29: INVOKEVIRTUAL
    binding/C#getValue(I)I
30: ISTORE 3 (v)
```

クラス C1 の getValue メソッド

```
0: (L705613182)
1: (line=19)
2: ILOAD 1 (x)
3: ICONST_1
4: IADD
5: IRETURN
```

クラス C2 の getValue メソッド

```
0: (L332181545)
1: (line=24)
2: ILOAD 1 (x)
3: ICONST_2
4: IADD
5: IRETURN
```

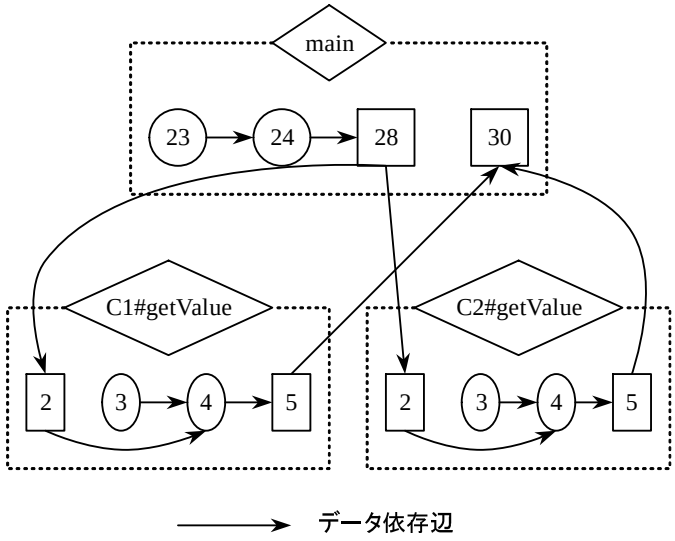

図 11: メソッドの呼び出し先の候補が複数ある場合の依存関係

3.6 SDG の作成

本研究で作成する SDG の例を図 12 に示す．図 12 は，図 4 のプログラムに対応するバイトコードの SDG である．各メソッドのバイトコードは図 5，9，13，14 の通りである．A#⟨init⟩ はクラス A のコンストラクタを表す．また，色付で示した頂点は，図 4 の 32 行目で使用している変数 id について後ろ向き Thin slicing を行う際に探索する頂点である．

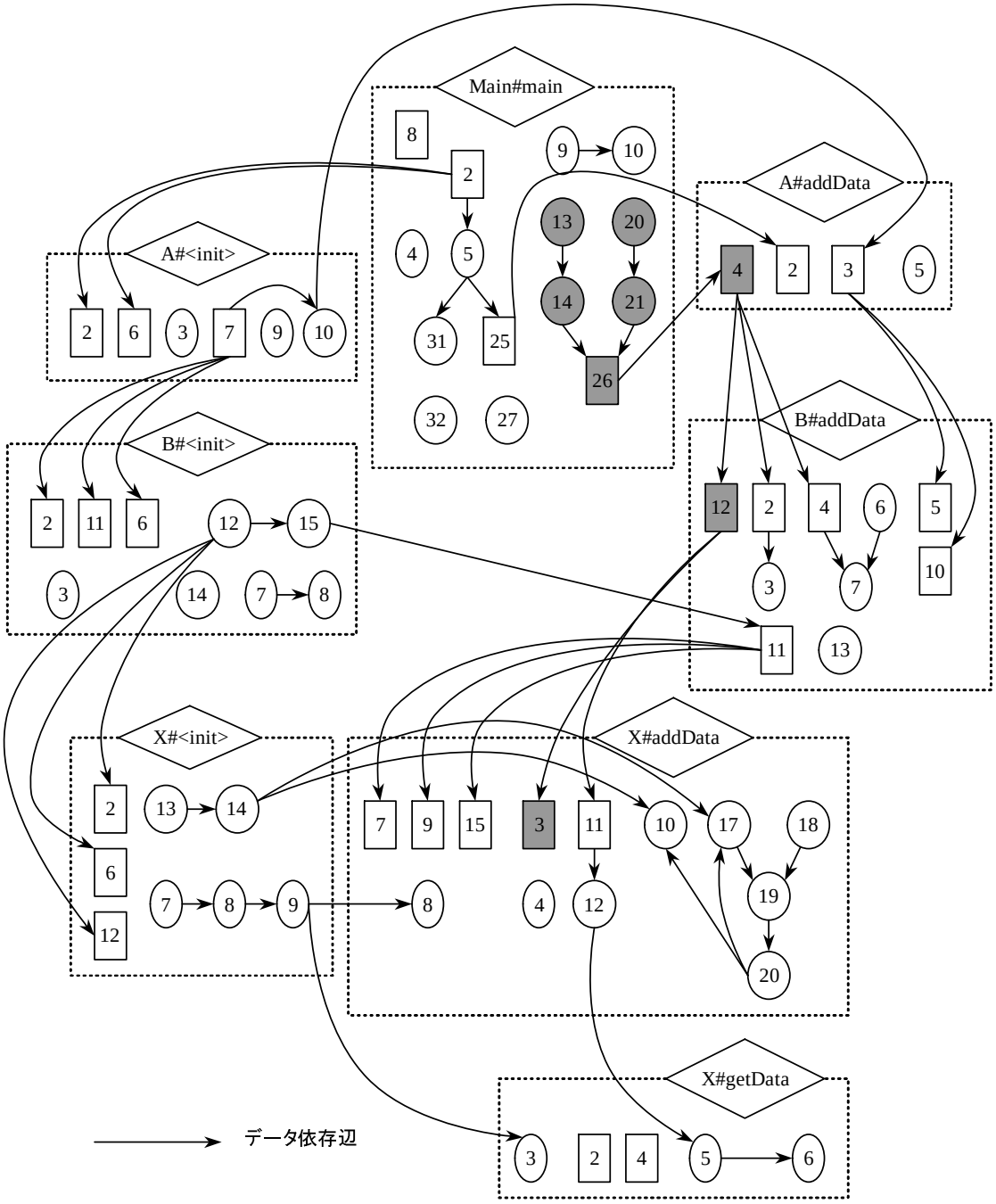

図 12: Thin slicing の SDG

クラス A のコンストラクタ

```
0: (L536111262)
1: (line=14)
2: ALOAD 0 (this)
3: INVOKESPECIAL
   java/lang/Object#<init>()V
4: (L1728081269)
5: (line=15)
6: ALOAD 0 (this)
7: NEW
8: DUP
9: INVOKESPECIAL sample/B#<init>()V
10: PUTFIELD sample/A#b: sample/B
11: (L1293046055)
12: (line=14)
13: RETURN
14: (L1838022392)
```

クラス A の addData メソッド

```
0: (L1728468445)
1: (line=17)
2: ALOAD 0 (this)
3: GETFIELD sample/A#b: sample/B
4: ILOAD 1 (id)
5: INVOKEVIRTUAL sample/B#addData(I)V
6: (L1357862146)
7: (line=18)
8: RETURN
9: (L280371153)
```

クラス X のコンストラクタ

```
0: (L332181545)
1: (line=28)
2: ALOAD 0 (this)
3: INVOKESPECIAL
   java/lang/Object#<init>()V
4: (L1665228262)
5: (line=29)
6: ALOAD 0 (this)
7: BIPUSH
8: NEWARRAY
9: PUTFIELD sample/X#idList: int[]
10: (L1484511730)
11: (line=30)
12: ALOAD 0 (this)
13: ICONST_0
14: PUTFIELD sample/X#count: int
15: (L723635264)
16: (line=28)
17: RETURN
18: (L937989087)
```

図 13: クラス A のメソッドとクラス X のコンストラクタのバイトコード

クラス B のコンストラクタ

```
0: (L1022165816)
1: (line=20)
2: ALOAD 0 (this)
3: INVOKESPECIAL
   java/lang/Object#<init>()V
4: (L1819177159)
5: (line=21)
6: ALOAD 0 (this)
7: ICONST_4
8: PUTFIELD sample/B#max: int
9: (L1185828974)
10: (line=22)
11: ALOAD 0 (this)
12: NEW
13: DUP
14: INVOKESPECIAL sample/X#<init>()V
15: PUTFIELD sample/B#x: sample/X
16: (L1579321858)
17: (line=20)
18: RETURN
19: (L764590486)
```

クラス B の addData メソッド

```
0: (L1867546546)
1: (line=24)
2: ILOAD 1 (id)
3: IFLT L233814070
4: ILOAD 1 (id)
5: ALOAD 0 (this)
6: GETFIELD sample/B#max: int
7: IF_ICMPGT L233814070
8: (L1965484127)
9: (line=25)
10: ALOAD 0 (this)
11: GETFIELD sample/B#x: sample/X
12: ILOAD 1 (id)
13: INVOKEVIRTUAL
      sample/X#addData(I)V
14: (L233814070)
15: (line=26)
16: FRAME-OP(3)
17: RETURN
18: (L1298264335)
```

図 14: クラス B のメソッドのバイトコード

3.7 用語の定義

3.7.1 バイトコードの分類

本研究では，バイトコードを以下の 3 つに分類する．

source 演算，メソッドあるいは定数によってデータを生成する命令．

sink 演算あるいはメソッドによってデータを使用する命令．

transfer source，sink のどちらでもない命令．

引数が 1 つ以上存在し，戻り値が使われているメソッド呼び出しと IADD などの演算は，1 つの命令で source と sink の 2 つに該当する．このような演算を行う命令を表 4 に示す．また，定数によってデータを生成する source に該当する命令を表 5，sink に該当する演算を行う命令を表 6，メソッドを呼び出す命令を表 7 に示す．

3.7.2 表記方法の定義

本研究で用いる Thin slice に関する表記を以下に定義する．

$Backward(v)$： ある sink v をスライシング基準とした後ろ向き Thin slice．また，その要素数を $|Backward(v)|$ で表す．

$Source(v)$： $Backward(v)$ のうち，source である Thin slice．また，その要素数を $|Source(v)|$ で表す．

$Forward(w)$： ある source w をスライシング基準とした前向き Thin slice．また，その要素数を $|Forward(w)|$ で表す．

$Sink(w)$： $Forward(w)$ のうち，sink である Thin slice．また，その要素数を $|Sink(w)|$ で表す．

$|Method(S)|$： Thin slice S がまたがるメソッド数．

$|Class(S)|$： Thin slice S がまたがるクラス数．

例として，図 7 のプログラムの場合，sink に対応する頂点は {6，14，18} であり，インデクス 18 をスライシング基準とした後ろ向き Thin slice を表す $Backward(18)$ は，インデクス 19 以外のすべての頂点となる．また，$Source(18)$ は {0，2，6，8，10，14} となる．

表 4: source かつ sink となる演算を行う命令

IADD	LADD	FADD	DADD	ISUB
LSUB	FSUB	DSUB	IMUL	LMUL
FMUL	DMUL	IDIV	LDIV	FDIV
DDIV	IREM	LREM	FREM	DREM
INEG	LNEG	FNEG	DNEG	ISHL
LSHL	ISHR	LSHR	IUSHR	LUSHR
IAND	LAND	IOR	LOR	IXOR
LXOR	IINC	I2L	I2F	I2D
L2I	L2F	L2D	F2I	F2L
F2D	D2I	D2L	D2F	I2B
I2C	I2S	LCMP	FCMPL	FCMPG
DCMPL	DCMPG			

表 5: 定数によってデータを生成する source となる命令

ACONST_NULL	ICONST_M1	ICONST_0	ICONST_1
ICONST_2	ICONST_3	ICONST_4	ICONST_5
LCONST_0	LCONST_1	FCONST_0	FCONST_1
FCONST_2	DCONST_0	DCONST_1	BIPUSH
SIPUSH	LDC		

表 6: sink となる演算を行う命令

IFEQ	IFNE	IFLT	IFGE
IFGT	IFLE	IF_ICMPEQ	IF_ICMPNE
IF_ICMPLT	IF_ICMPGE	IF_ICMPGT	IF_ICMPLE
IF_ACMPEQ	IF_ACMPNE		

表 7: メソッドを呼び出す命令

INVOKEVIRTUAL	INVOKEINTERFACE
INVOKESPECIAL	INVOKESTATIC

4　評価実験

Thin slice のプログラム理解における有用性を評価する実験を行う．本実験は，Binkley らの実験 [3] を参考にしている．Binkley らの実験では，後ろ向きスライスと前向きスライスの SDG における頂点数をそれぞれ計算し，その平均値を求めることによって，スライスサイズの期待値を計算している．そこで，本実験でも後ろ向き Thin slice と前向き Thin slice の SDG における頂点数を計算する．

本実験のリサーチクエスチョンは以下の 2 つである．

RQ1　Thin slice のサイズは，平均的に十分小さいものであるか.

RQ2　Thin slice はプログラム理解において，どれくらい効果の高いものであるか.

以降では，実験方法の詳細と実験結果について述べる．

4.1　実験方法

本実験では，実験対象の Java プログラムの class ファイルからバイトコードを取得し，Thin slice を計算する．その計算結果から各 RQ に対応した指標を計測する．

4.1.1　RQ1 に対する指標

RQ1 について調査するために，すべての sink v について $|Backward(v)|$ を計測し，すべての source w について $|Forward(w)|$ を計測する．$|Backward(v)|$ は，v を基準とした後ろ向き Thin slice の SDG における頂点数を計算することによって求める．また，$|Forward(w)|$ は，w を基準とした前向き Thin slice の SDG における頂点数を計算することによって求める．これらの平均値が十分に小さければ，Thin slice のサイズは平均的に十分小さいと言える．

4.1.2　RQ2 に対する指標

RQ2 について調査するために，すべての sink v について次の指標を計測する．

- $|Method(Backward(v))|$
- $|Class(Backward(v))|$
- $|Source(v)|$
- $|Method(Source(v))|$

- $|Class(Source(v))|$

また，すべての source w について次の指標を計測する．

- $|Method(Forward(w))|$
- $|Class(Forward(w))|$
- $|Sink(w)|$
- $|Method(Sink(w))|$
- $|Class(Sink(w))|$

を計測する．

$|Source(v)|$ は，v を基準とした後ろ向き Thin slice に含まれる命令のうち，3.7.1 項で述べた source に該当する命令の頂点数を計算することによって求める．また，$|Sink(w)|$ は，w を基準とした前向き Thin slice に含まれる命令のうち，sink に該当する命令の頂点数を計算することによって求める．$|Source(v)|$ の値が小さい場合は Thin slice を利用することで，使用しているデータの生成元を少数に特定できる．同様に，$|Sink(w)|$ の値が小さい場合は生成したデータの使用先を少数に特定できる．

$|Method(Backward(v))|$, $|Class(Backward(v))|$, $|Method(Forward(w))|$, $|Class(Forward(w))|$ については，これらの値が大きい場合，様々なメソッドやクラスにまたがった Thin slice であることを意味するため，Thin slice によってメソッド呼び出しをたどってデータの生成元や使用先を特定する作業の時間を減らすことが期待できる．

$|Method(Source(v))|$, $|Method(Sink(w))|$ については，$|Method(Backward(v))|$, $|Method(Forward(w))|$ との値の差が小さい場合，ある 1 つのメソッド内で同じ値が伝わっていくデータフローが少ないと考えられるため，データを伝播しているだけの経路をたどってデータの生成元や使用先を特定する作業を Thin slice によって省略することは期待できない．$|Class(Source(v))|$, $|Class(Sink(w))|$ と $|Class(Backward(v))|$, $|Class(Forward(w))|$ についても同様である．

4.2 実験対象

実験対象のプログラムは DaCapo benchmark（9.12）[1] とした．DaCapo benchmark には，Java 言語で書かれた多数のアプリケーションが含まれている．本実験で対象としたプログラムの規模は表 8 の通りである．

4.3 実験結果と考察

本実験で計測した各指標の最大値と平均値，また $|Backward(v)|$ と $|Forward(w)|$ について全頂点に占める最大値と平均値の割合を表 9 に示す．また，各指標の分布を図 15 から図 26 に示す．図 15 から図 26 は各指標の値を横軸とし，その値以下である sink（source）が全 sink（source）に占める割合を縦軸に示している．

表 9 から $|Backward(v)|$ と $|Forward(w)|$ の平均値は，7 つのプログラムで平均して約 1.4 %であり，従来のスライシングでは約 30 %であるのに対して，十分小さい値であることが分かる．この結果から RQ1 について，Thin slice のサイズは平均的に十分小さいと言える．

各指標の分布については，図 17 と図 18，また図 19 と図 20 から，$Backward(v)$ と $Forward(w)$ を合わせて，全 Thin slice 中の約 75 %の Thin slice が複数のメソッドにまたがり，約 55 %の Thin slice が複数のクラスにまたがることが分かる．図 21 と図 22 から，全 sink（source）中の約 50 %の sink v（source w）について，$|Source(v)|$（$|Sink(w)|$）の値が 6 以下であることが分かる．$|Source(v)|$ の値が 6 以下である sink v について，$|Method(Backward(v))|$ の分布を求めた結果が図 27 である．図 27 から，後ろ向き Thin slice が複数のメソッドにまたがり，かつ $|Source(v)|$ の値が 6 以下である sink が全 sink の約 30 %を占めることが分かる．図 17 と図 23 から，$|Method(Backward(v))|$ と $|Method(Source(v))|$ の各値における割合の差は，メソッド数 1 の場合が約 15 %で以後小さくなっていることが分かる．$|Method(Forward(w))|$ と $|Method(Sink(w))|$ についても図 18 と図 25 から分布の差が小さいことが分かる．図 19，24，20，26 から $|Class(Backward(v))|$ と $|Class(Source(v))|$，また $|Class(Forward(w))|$, $|Class(Sink(w))|$ も同様である．図 28 は，全頂点数に占める $|Backward(v)|$ の割合を横軸とし，全 sink に占める割合を縦軸とした度数分布である．図 28 から，全 sink の 80 %以上の $|Backward(v)|$ が非常に小さい値であるが，約 10〜20 %の

表 8: 実験対象のプログラム

プログラム名	クラス数	メソッド数	SDG 上の頂点数
tomcat	261	2,389	54,468
luindex	560	4,180	123,191
sunflow	657	4,609	190,526
avrora	1,838	9,304	211,343
pmd	2,369	16,439	448,722
xalan	2,805	22,377	815,861
batik	4,417	28,818	968,470

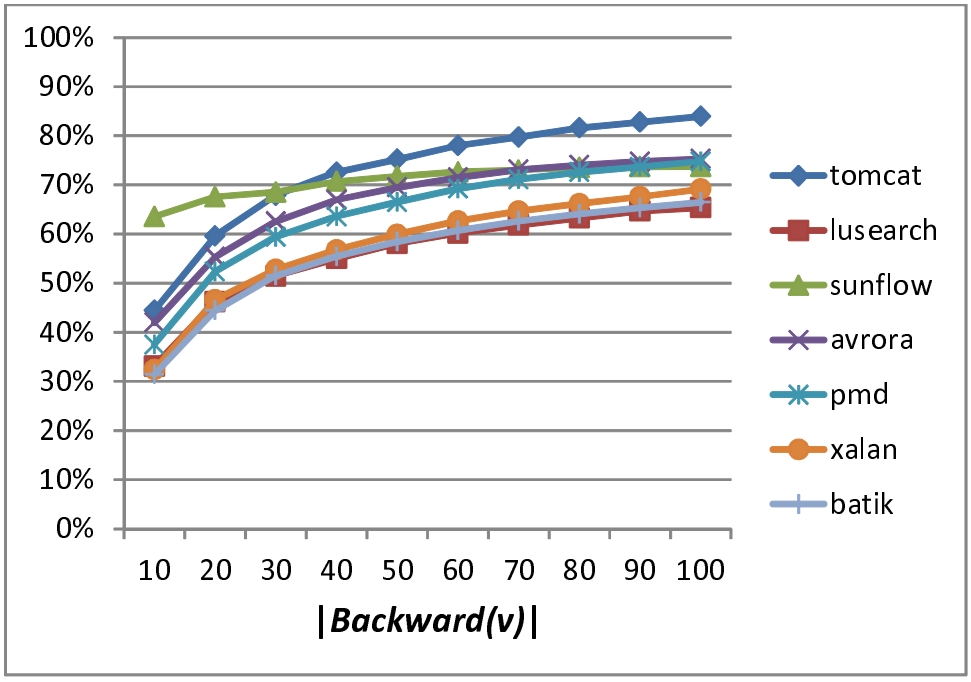

図 15: $|Backward(v)|$ の累積度数分布

$|Backward(v)|$ が各プログラムの $|Backward(v)|$ の最大値付近に分布しており，非常に大きな値となっている．

以上の結果から RQ2 について，プログラム理解において変数の値を調査したい場合，すべての後ろ向き Thin slice と前向き Thin slice の約 75 %で複数のメソッドにまたがるデータフローを表現した Thin slice となることから，メソッド呼び出しをたどってデータの生成元を特定する作業の簡略化が期待できる．特に，全後ろ向き Thin slice 中の約 30 %の Thin slice は，データの生成元を少数に特定することも可能である．しかし，$|Method(Backward(v))|$ と $|Method(Source(v))|$，あるいは $|Method(Forward(w))|$ と $|Method(Sink(w))|$ の分布の差が小さいことから，ある 1 つのメソッド内で同じ値を伝播していくデータフローは少ないと考えられる．そのため，少数のメソッドで多くの文をたどるような場面における効果は期待できない．また，すべての後ろ向き Thin slice の約 10~20 %は，スライスサイズが非常に大きくなってしまい，プログラム理解における効果は期待できない．

表 9: 各指標の統計量

		tomcat	luindex	sunflow	avrora
全頂点数		54,468	123,191	190,526	211,343
$\|Backward(v)\|$	最大値	1,094	10,502	19,415	9,705
	最大値の割合（%）	2.0	8.5	10.2	4.6
	平均値	64.6	1619.2	4098.3	731.6
	平均値の割合（%）	0.12	1.3	2.2	0.35
$\|Forward(w)\|$	最大値	3,673	23,633	43,041	27,592
	最大値の割合（%）	6.7	19.2	22.6	13.1
	平均値	57.2	1434.0	7834.8	2386.0
	平均値の割合（%）	0.11	1.2	4.1	1.1
$\|Method(Backward(v))\|$	最大値	92	1,088	1,099	1,473
	平均値	8.0	155.6	155.7	128.7
$\|Method(Forward(w))\|$	最大値	368	1,852	2,449	3,702
	平均値	6.4	116.1	230.2	331.0
$\|Class(Backward(v))\|$	最大値	41	267	220	510
	平均値	2.3	39.6	35.6	51.8
$\|Class(Forward(w))\|$	最大値	68	375	348	1,147
	平均値	2.3	35.9	46.2	98.9
$\|Source(v)\|$	最大値	458	3,948	8,276	3,943
	平均値	25.7	650.4	1861.4	273.7
$\|Sink(w)\|$	最大値	1,610	9,216	17,369	11,689
	平均値	25.4	537.6	3031.9	1021.4
$\|Method(Source(v))\|$	最大値	71	896	894	1,173
	平均値	6.5	131.1	130.6	97.8
$\|Class(Source(v))\|$	最大値	41	254	170	401
	平均値	2.1	37.7	33.1	38.7
$\|Method(Sink(w))\|$	最大値	321	1,581	2,208	3,372
	平均値	5.5	99.1	191.4	299.8
$\|Class(Sink(w))\|$	最大値	67	359	341	1,098
	平均値	2.1	33.1	44.7	91.9

表 9 の続き

		pmd	xalan	batik
全頂点数		448,722	815,861	968,470
$\|Backward(v)\|$	最大値	14,869	34,089	60,772
	最大値の割合（%）	3.3	4.2	6.3
	平均値	1505.8	4792.2	10814.4
	平均値の割合（%）	0.34	0.59	1.1
$\|Forward(w)\|$	最大値	52,339	105,749	140,965
	最大値の割合（%）	11.7	13.0	14.7
	平均値	6226.3	18481.9	28719.3
	平均値の割合（%）	1.4	2.3	3.0
$\|Method(Backward(v))\|$	最大値	1,265	2,864	3,427
	平均値	126.5	385.5	576.9
$\|Method(Forward(w))\|$	最大値	3,132	7,175	8,011
	平均値	369.6	1267.1	1632.0
$\|Class(Backward(v))\|$	最大値	297	797	974
	平均値	31.2	113.0	170.1
$\|Class(Forward(w))\|$	最大値	507	1,306	1,750
	平均値	67.0	243.4	361.6
$\|Source(v)\|$	最大値	5,723	13,669	24,695
	平均値	542.0	1789.8	4458.6
$\|Sink(w)\|$	最大値	19,425	41,423	52,334
	平均値	2264.2	7238.5	10662.3
$\|Method(Source(v))\|$	最大値	1,047	2,187	2,694
	平均値	108.2	298.6	457.8
$\|Class(Source(v))\|$	最大値	271	671	872
	平均値	28,4	92.7	150.8
$\|Method(Sink(w))\|$	最大値	2,535	6,045	6,563
	平均値	300.0	1071.1	1336.6
$\|Class(Sink(w))\|$	最大値	473	1,243	1,677
	平均値	62.0	231.1	345.7

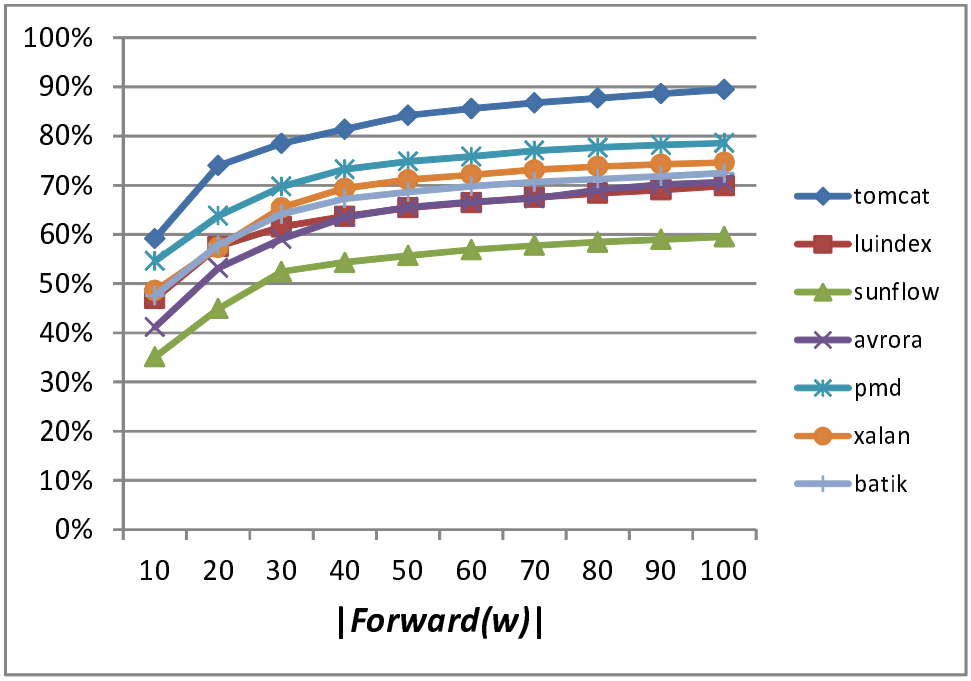

図 16: $|Forward(w)|$ の累積度数分布

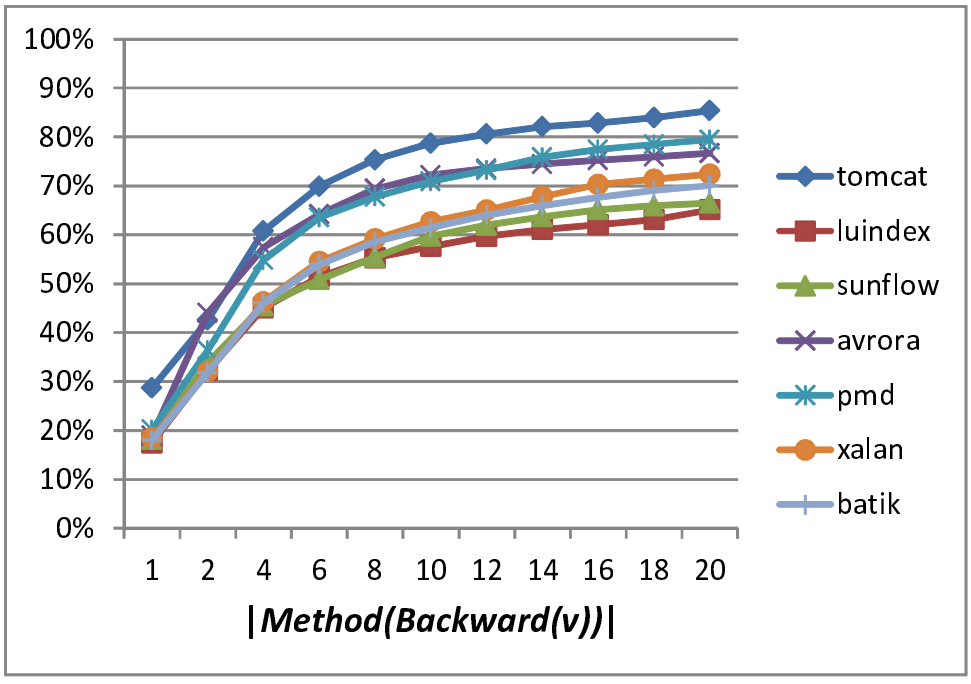

図 17: $|Method(Backward(v))|$ の累積度数分布

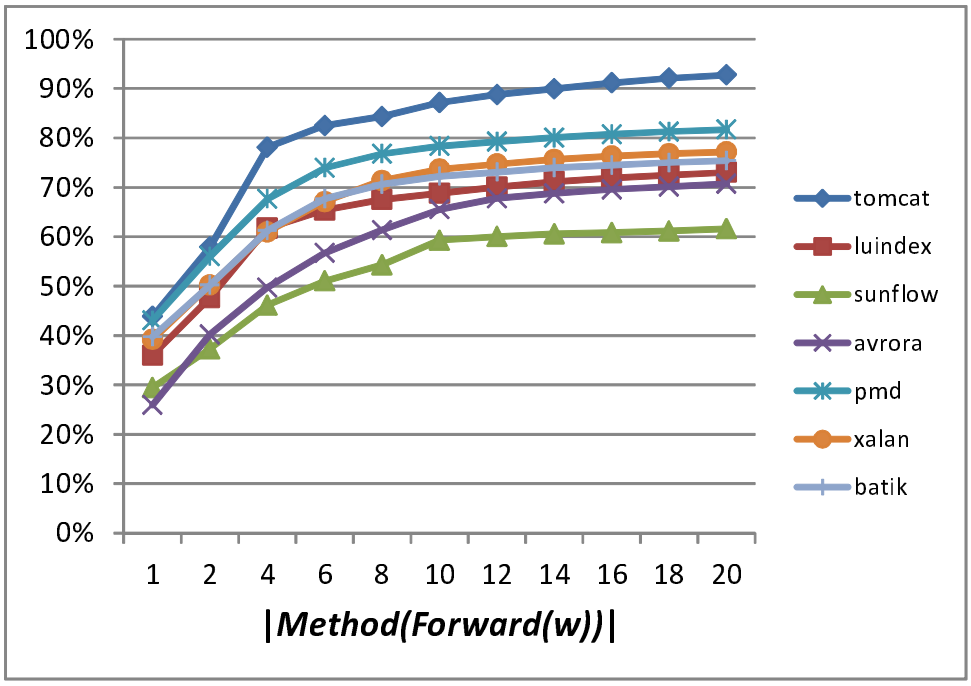

図 18: $|Method(Forward(w))|$ の累積度数分布

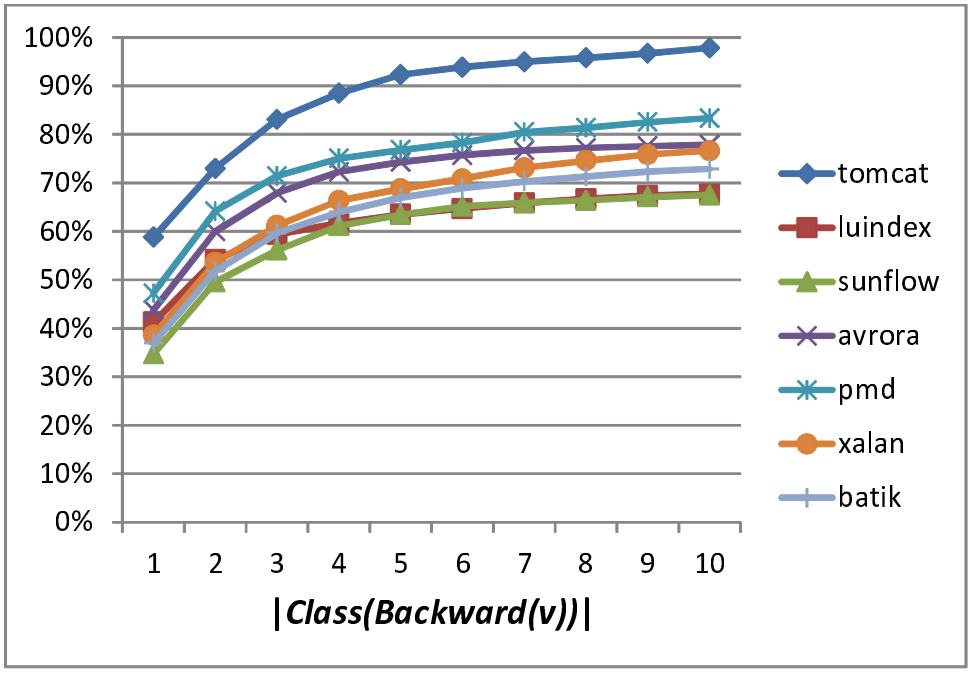

図 19: $|Class(Backward(v))|$ の累積度数分布

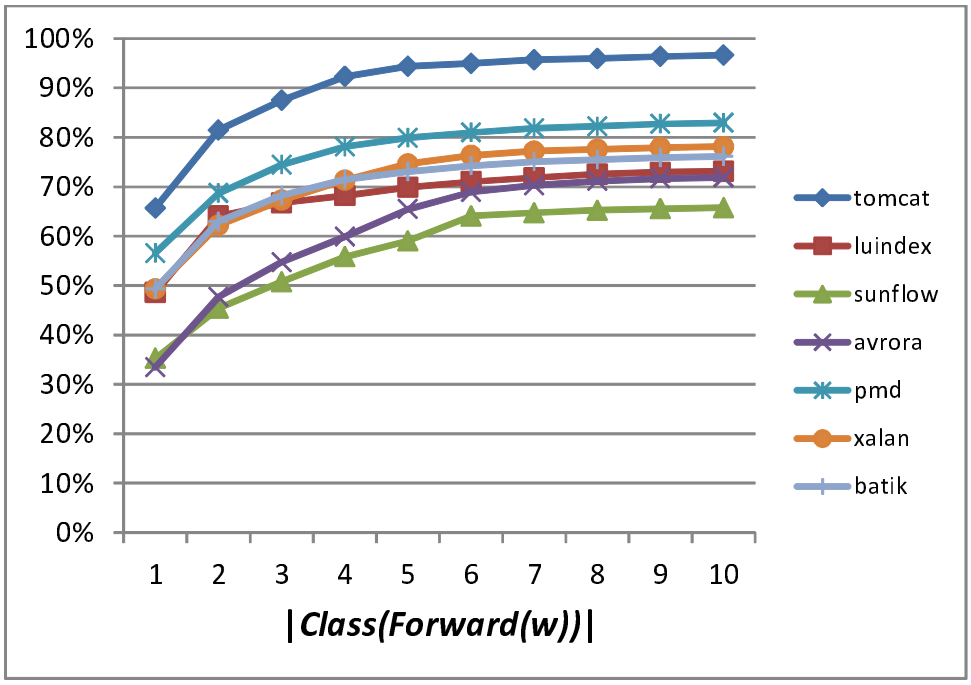

図 20: $|Class(Forward(w))|$ の累積度数分布

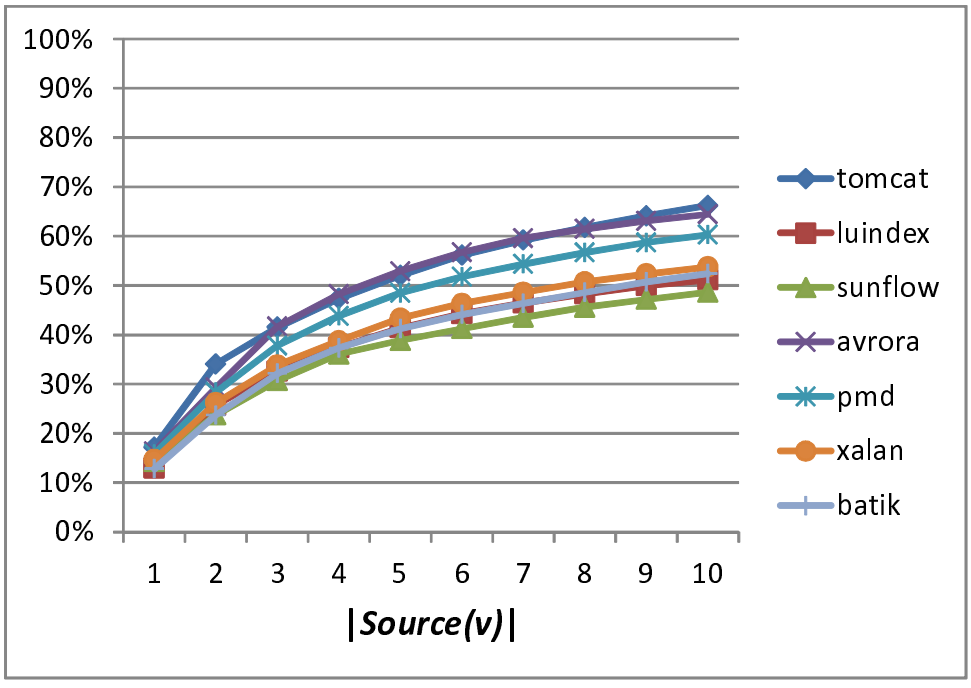

図 21: $|Source(v)|$ の累積度数分布

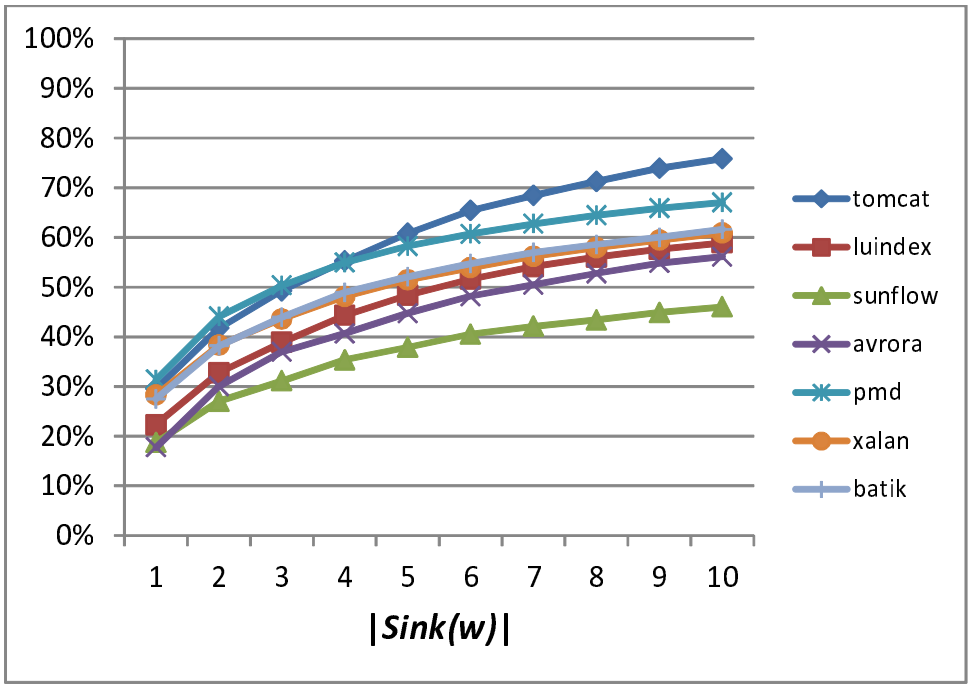

図 22: $|Sink(w)|$ の累積度数分布

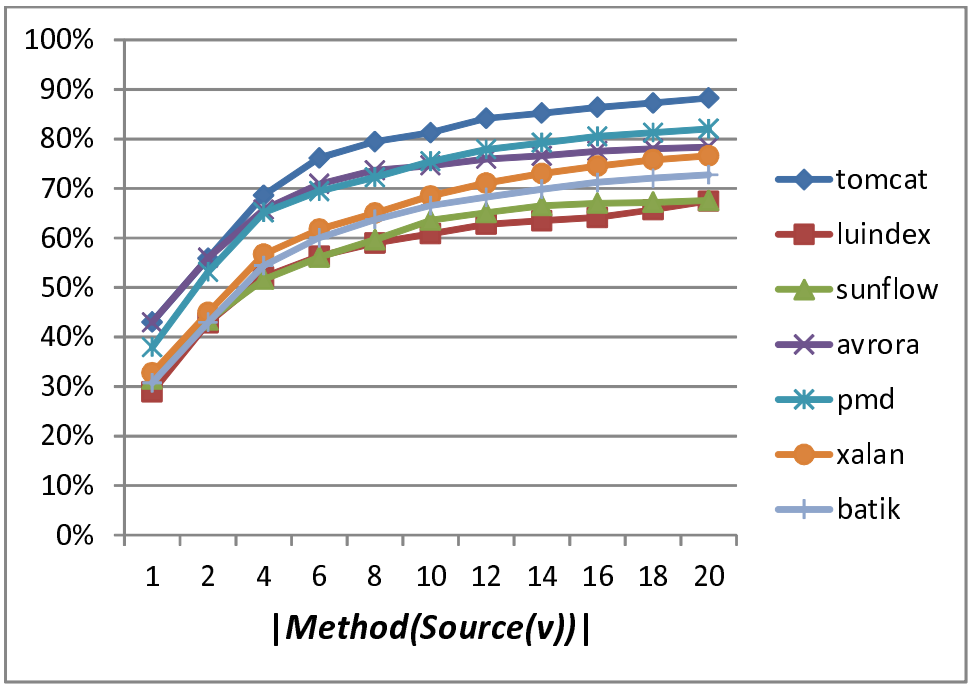

図 23: $|Method(Source(v))|$ の累積度数分布

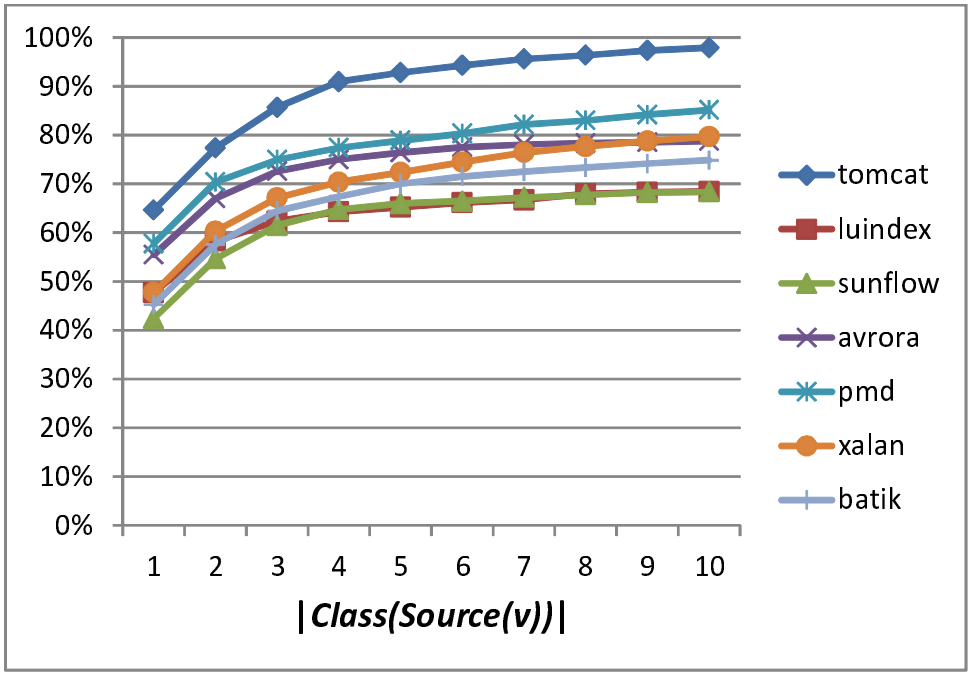

図 24: $|Class(Source(v))|$ の累積度数分布

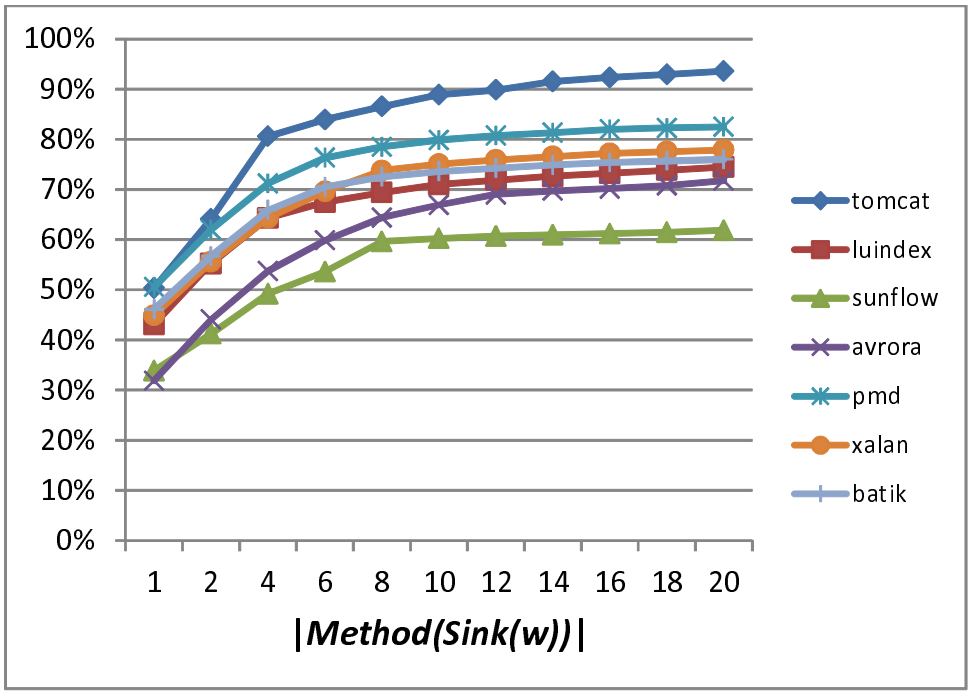

図 25: $|Method(Sink(w))|$ の累積度数分布

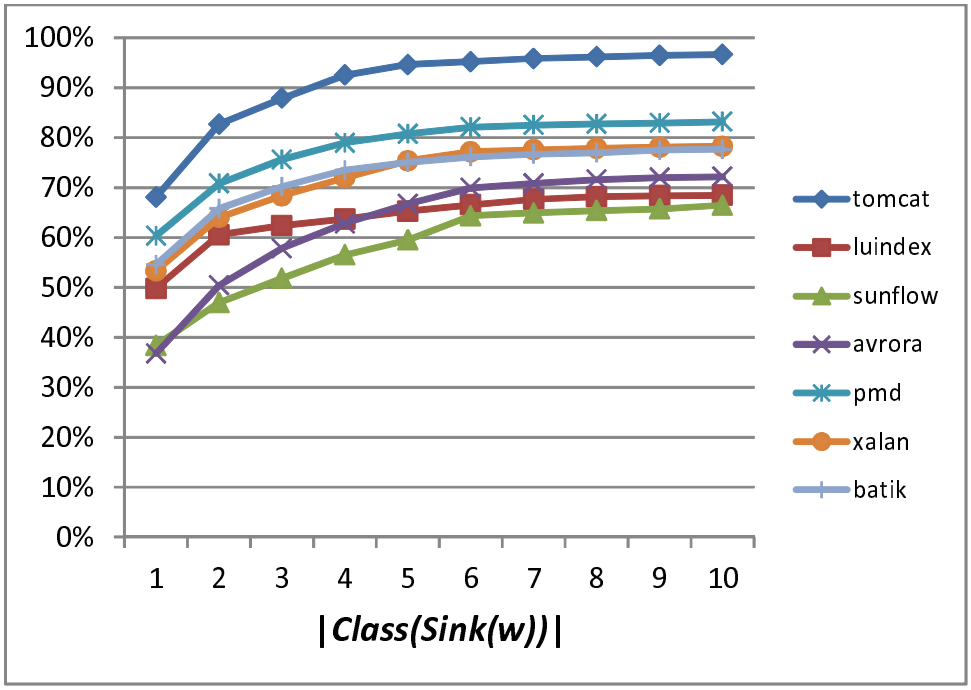

図 26: $|Class(Sink(w))|$ の累積度数分布

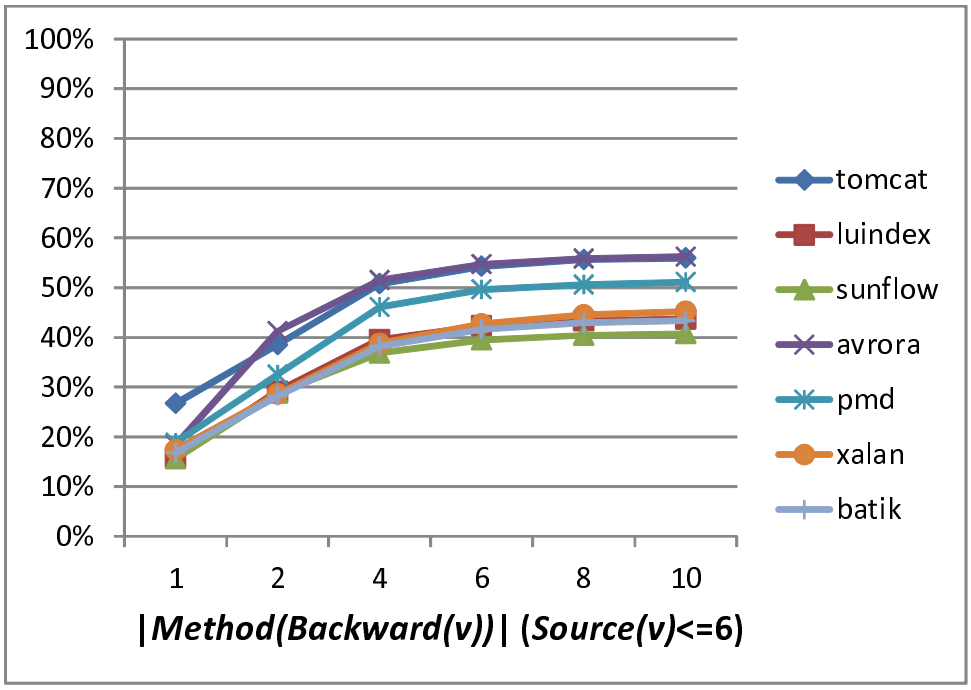

図 27: $|Source(v)|$ が 6 以下である sink v の $|Method(Backward(v))|$ の累積度数分布

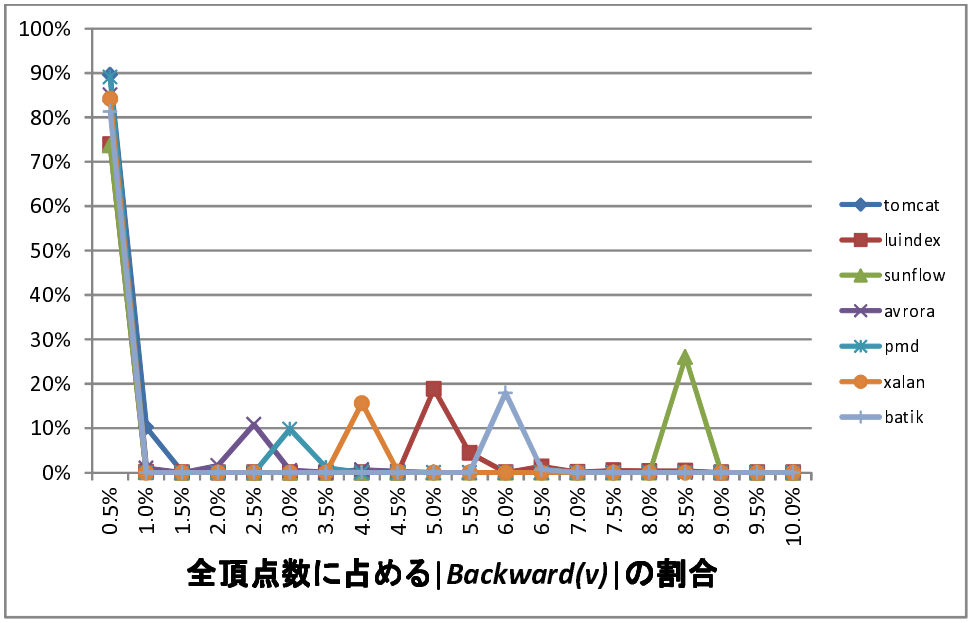

図 28: 全頂点数に占める $|Backward(v)|$ の割合の度数分布

5 関連研究

スライスが開発者に与える影響について多くの実験が行われている [9, 11, 16]．Ishio ら [9] は，データフローを可視化することによって，開発者がプログラムの調査を早く行えるようになるということを示している．Kusumoto ら [11] は，スライスを利用する場合の方が利用しない場合よりデバッグ作業の時間が早くなるということを示している．Sridharan ら [16] は，Thin slice を利用するとデバッグ作業やプログラム理解にかかる時間が減少することを示している．これらの実験から，開発者がデバッグ作業やプログラム理解を行う際に，データフロー情報を提示することが有用であると考えられる．

また，スライスに関する数値指標について様々な調査が行われている [3, 4, 10]．Binkley らは，スライスサイズの分布を調査しており [3]，スライスサイズの分布を利用して，プログラムの構造の複雑さを評価する手法を提案している [4]．Jász ら [10] は，スライスの高速な近似計算の結果がスライスサイズに対してどの程度大きいかということを調査している．

6　まとめと今後の課題

本研究では，Thin slice の統計的調査を行うことによって，プログラム理解における Thin slice の有用性を評価した．評価実験では，DaCapo benchmark の 7 個の Java プログラムに対して Thin slice サイズや Thin slice がまたがるメソッド数といった指標を計測した．その結果，Thin slice のサイズは平均的に十分小さくなることが確認できた．また，プログラム中のデータ使用場所や生成場所の約 75 %で，メソッド呼び出しをたどってデータの生成元を特定する作業の簡略化が期待できることが分かった．さらに，データ使用場所の約 30 %で，メソッド呼び出しをたどる必要のあるデータについて，そのデータの生成元を少数に特定することが可能であると考えられる．

今後の課題については，Thin slice によるデータフローのより詳細な調査が考えられる．例えば，複数の sink に対する後ろ向き Thin slice を計算した結果，そのデータフローに共通の経路が存在する場合に，それらの source で生成されたデータはプログラムにおいて同様の役割を担っている可能性が高い．このような情報をプログラム理解に利用することが考えられる．また，Thin slice の結果をプログラム理解支援ツールに利用することが考えられる．具体的には，鹿島らのツール [21] と組み合わせることにより，メソッドの入力データとして到達する値を可視化することが考えられる．

謝辞

本研究において，常に適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝いたします.

本研究において，随時適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授に深く感謝いたします.

本研究において，終始適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻石尾隆助教に深く感謝いたします.

本研究において，適時適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻眞鍋雄貴特任助教に深く感謝いたします.

本研究において，様々な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻鹿島悠氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] Dacapo. `http://dacapobench.org/`.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[3] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, Vol. 16, No. 2, pp. 1–32, 2007.

[4] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 177–186, 2005.

[5] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, Vol. 28, No. 2, pp. 294–306, 1989.

[6] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. *Tutorial on Software Maintenance, IEEE Computer Society press*, pp. 13–27, 1983.

[7] C. Hammer and G. Snelting. An improved slicer for java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 17–22, 2004.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 35–46, 1988.

[9] T. Ishio, S. Etsuda, and K. Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading. In *Proceedings of the 20th International Conference on Program Comprehension*, pp. 37–46, 2012.

[10] J. Jász, A. Beszédes, T. Gyimóthy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the 24th International Conference on Software Maintenance*, pp. 137–146, 2008.

[11] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, Vol. 7, No. 1, pp. 49–76, 2002.

[12] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 185–194, 2010.

[13] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 492–501, 2006.

[14] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 1, pp. 1–41, 2005.

[15] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 5, pp. 11–20, 1994.

[16] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 112–122, 2007.

[17] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, systems, languages, and applications*, pp. 264–280, 2000.

[18] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 87–97, 2009.

[19] J. Wang, X. Peng, Z. Xing, and W. Zhao. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pp. 213–222, 2011.

[20] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[21] 鹿島悠, 石尾隆, 井上克郎. エイリアス解析を用いたメソッドの入力データの利用法可視化ツール. ソフトウェアエンジニアリングシンポジウム 2012 論文集, pp1-8, 2012.