特別研究報告

題目

バイトコードを単位とする Java スライスシステムの試作

指導教官
井上克郎 教授

報告者
梅森 文彰

平成 14 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 13 年度 特別研究報告

バイトコードを単位とする Java スライスシステムの試作

梅森 文彰

内容梗概

プログラムのデバッグを効率よく行なう手法の一つに，プログラムスライス (以下，スライス) がある．スライスとは，プログラム中のある文のある変数に着目した時，その変数に影響を与える可能性のある文の集合である．スライスはプログラム文間の依存関係解析により選られ，主なものに，静的スライスと動的スライスがある．

また，近年のソフトウェア開発環境におけるオブジェクト指向言語の利用の高まりに伴い，オブジェクト指向プログラムを対象とするスライスシステムが求められるようになった．しかし，オブジェクト指向言語には，従来の手続き型言語と比べ多くの実行時決定要素が存在し，実利用を考慮したスライスシステムの実現はなされていなかった．

そこで本研究では，オブジェクト指向言語 Java で記述されたプログラムを対象とするスライスシステムの試作を行う．システムの実現にあたり，静的スライスと動的スライスの中間に位置する DC スライスを採用することで，高精度のスライスを低コストで抽出することができる．加えて，バイトコードを単位とする依存関係解析を行っており，さらなる精度向上も期待できる．

今回試作したシステムは，ソースコード・バイトコード対応表出力を行う Java コンパイラ，バイトコード間の動的データ依存関係解析を行う Java バーチャルマシン，バイトコードを対象とする静的制御依存関係解析ツール，プログラム依存グラフに基づくスライサで構成される．また実際に，いくつかの Java プログラムを適用し，システムの有効性を確認した．

主な用語

Java
Java Compiler
プログラムスライス
動的解析
静的解析

目 次

1 まえがき **3**

2 プログラムスライス **5**

2.1 プログラムスライス計算法 5

2.2 静的スライス 7

2.3 動的スライス 7

2.4 Dependence Cache (DC) スライス 7

2.5 各スライス手法の比較 14

3 DC スライスの Java への適用 **16**

3.1 概要 16

3.2 解析手順 16

3.3 Phase1: ソースコード・バイトコード対応表の出力 17

3.4 Phase2(a): 静的制御依存関係解析 21

3.5 Phase3: PDG によるスライス計算 32

4 提案手法の実現 **35**

4.1 システム構成 35

4.2 評価 36

4.2.1 スライスサイズ 37

4.2.2 解析コスト 38

5 むすび **42**

謝辞 **43**

参考文献 **44**

1　まえがき

プログラムデバッグを効率よく行う手法の一つに，プログラムスライス(*Program Slice*，以下，スライス) がある．プログラムスライシング (*Program Slicing*) とは，プログラム中のある文 s のある変数 v (スライス基準 (*Slicing Criterion*) $<s, v>$ と呼ぶ) に対し，v の値に影響を与える可能性のあるすべての文をプログラムから抽出する技法で，Weiser[17] によって提案された．一般に，スライスはプログラム文間の依存関係 (*Dependence Relation*) の解析により得られる．

静的スライス (*Static Slice*) [17] とは，ソースコードに対し，起こり得る全ての実行経路を考慮して依存関係解析を行うことにより得られるスライスで，小さなコストでの解析が可能であるが，実行に関係しない文 (デバッグ時に必要ないと考えられる文) もスライスに含まれるという欠点がある．

これに対して動的スライス(*Dynamic Slice*)[1] とは，実際に実行された実行系列(*Execution Trace*) に対して依存関係解析を行うことにより計算されるスライスである．静的スライスと比較し，スライスサイズの減少 (精度の向上) が期待できるが，実行系列の保存と解析には多くの空間，時間コストを必要となる．

DC スライス (*DC Slice*) [4][7] は，データ依存関係解析を動的に行うことにより静的スライスより高い精度のスライス計算が可能となり，また，実行系列を保存しないため，動的スライスに比べ非常に小さな解析コストでのスライス計算が可能となる．

また，近年のソフトウェア開発環境において，C などの手続き型言語だけでなく，Java や C++など，いわゆるオブジェクト指向言語の利用が高まっている．そこで，クラスや継承など，オブジェクト指向言語特有の概念を考慮したスライスシステムの構築が求められてきた．オブジェクト指向言語に対し，静的スライス [10]，動的スライス [18]，DC スライス [12] はそれぞれ提案されているが，実用的なシステムは実現されていない．

そこで，本研究ではオブジェクト指向言語である Java を対象としたスライスシステムの構築を行う．オブジェクト指向言語には，多くの実行時決定要素が含まれるため，システム構築を行うにあたり，静的制御依存関係解析と動的データ依存関係解析を組み合わせた DC スライスを採用することで，高精度のスライスを低コストで抽出することが可能となる．加えて，バイトコードを単位とする依存関係解析を行っており，さらなる精度向上も期待できる．

試作したシステムは，バイトコード・ソースコード対応表出力を行う **Java** コンパイラ (*Java Compiler*)，バイトコード間の動的データ依存関係解析を行う **Java** バーチャルマシン (*Java Virtual Machine*，以下 JVM)，バイトコードを対象とする静的制御依存関係解析ツール，プログラム依存グラフ (*Program Dependence Graph*，以下 PDG) に基づくス

ライサで構成される．バイトコードに対して計算された DC スライスを，コンパイラによって出力された対応表を利用することでソースコードに対応付ける．

以降，2 節で既存のプログラムスライス計算手法について述べ，3 節で DC スライスの Java への適用について述べる．次に，4 節で Java を対象とするスライスシステムを実装し，その評価を行う．最後に，5 節でまとめと今後の課題について述べる．

2　プログラムスライス

プログラムスライシング (*Program Slicing*) 技術とは，プログラム中のある文 s におけるある変数 v (スライス基準<s,v> と呼ぶ) に対して v の値に影響を与える全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．ある値 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，プログラム理解等にも利用される．

2.1　プログラムスライス計算法

スライスの計算にはさまざま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 [13] ．以降，いくつかの諸定義をしたのち，手法 [13] によるスライス計算手順について述べる．

プログラム文間に存在する依存関係

プログラム中の文の間には，以下に定義する制御依存関係，データ依存関係の 2 種類の依存関係が存在する．

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) 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) 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 の実行可能な経路で，その経路において s から t 間に変数 v を再定義している文が存在しない，というものが存在する

プログラム依存グラフ

プログラムに対して上述の依存関係の解析を行うことによって得られた依存関係情報は，プログラム依存グラフ (*Program Dependence Graph* . 以降，PDG) として出力される . PDG とは，プログラム内の文間の依存関係を表す有向グラフであり，その節点は，プログラムに含まれる条件判定部分，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及び特殊辺も存在する [16] .

プログラムスライス計算手順

上述に示した 2 つの依存関係とプログラム依存グラフの概念を用い，以下の手順でスライスの計算がなされる .

スライス計算法

Phase 1: 依存関係解析

各プログラム文に対し，以下の依存関係解析を行う .

(a) 制御依存関係解析

(b) データ依存関係解析

Phase 2: プログラム依存グラフ構築

Phase 1 で求めた依存関係を利用し，PDG を構築する .

Phase 3: スライス計算

スライス基準 $<s, v>$ に対するスライスを計算する . スライス基準 $<s, v>$ に対するスライスは，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によって計算される .

PDG や計算されるスライスの具体的例は，2.2. ，2.3. および 2.4. で紹介する .

スライスには，依存関係解析手法の違いにより，静的スライスと動的スライスの 2 種類に大別される . 静的スライスは，プログラムに対して，起こり得る全ての可能な入力データを考慮したスライスであり，動的スライスは，ある特定の入力のもとで生成される実行系列のみを解析して求められたスライスである . また、静的スライス・動的スライスを組み合わせたスライスとして，DC スライスがある .

以降，静的スライス，動的スライスおよび DC スライスについて，順に述べる .

2.2 静的スライス

静的スライスは，スライス計算の Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに静的に行うスライス計算手法である．与えられたソースコードの各文を節点とし，起こり得るすべての実行経路に対して依存関係解析を行う．

静的スライスは，現実には短い時間で計算される．静的スライスは，プログラムに起こり得るすべての実行経路を考慮して PDG を構築するため，プログラムに存在する特定の機能を抽出したい場合には有効である．しかし，プログラム実行においてすべての実行経路が利用されることは少なく，実行時エラーの原因を把握するためのフォールト位置特定に対しては効果的とはいえない．

図 1 の C 言語で記述されたソースコードに対し，静的スライスを計算するときに構築される PDG は図 2 のようになる．さらに，図 1 のソースコードの，スライス基準 $< 37, d >$ に関する静的スライスを図 3 に，比較のため元のソースプログラムと併せて示す．

2.3 動的スライス

動的スライスは，スライス計算の Phase 1 において (a) 制御依存関係解析，(b) データ依存関係解析をともに動的に行うスライス計算手法である．まず，特定の入力を与えてプログラムを実行する．そして，得られた実行系列の各実行時点 (*Execution Point*) を節点とし，特定の実行経路に対する依存関係解析による PDG を構築し，スライスを計算する．

動的スライスでは，解析対象を特定の実行経路に限定し，その実行の際に発生する依存関係に基づくものであるため，一般に計算されるスライスは静的スライスに比べて小さくなる．また，実際に実行された部分の中からのみスライスが計算されるため，フォールト位置特定を効率よく行うことができる．しかし，動的スライスの計算には，実行系列および，実行中に発生するすべての制御依存関係とデータ依存関係を記憶しなければならないため，多大な空間コストと時間コストを要する．とりわけ，実行系列の大きさはプログラム実行文の数に比例することから，入力データによっては非常に大きなものとなり，それに伴いスライス計算に要する時間も増大する．

図 1 の C 言語で記述されたソースコードに対し，入力として 2 を与えた場合の実行系列を図 5 に示す．また，構築される PDG を図 4 に示す．さらに，スライス基準 $< 37, d >$ に関する動的スライスを図 6 に，比較のため元のソースプログラムと併せて示す．

2.4 Dependence Cache (DC) スライス

例えば，配列を含むプログラムに対して静的スライスを計算する場合，配列の添字の値を静的に把握することは難しく，可能性のある添字値をすべて考慮しなければならないため不

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

図 1: サンプルソースコード

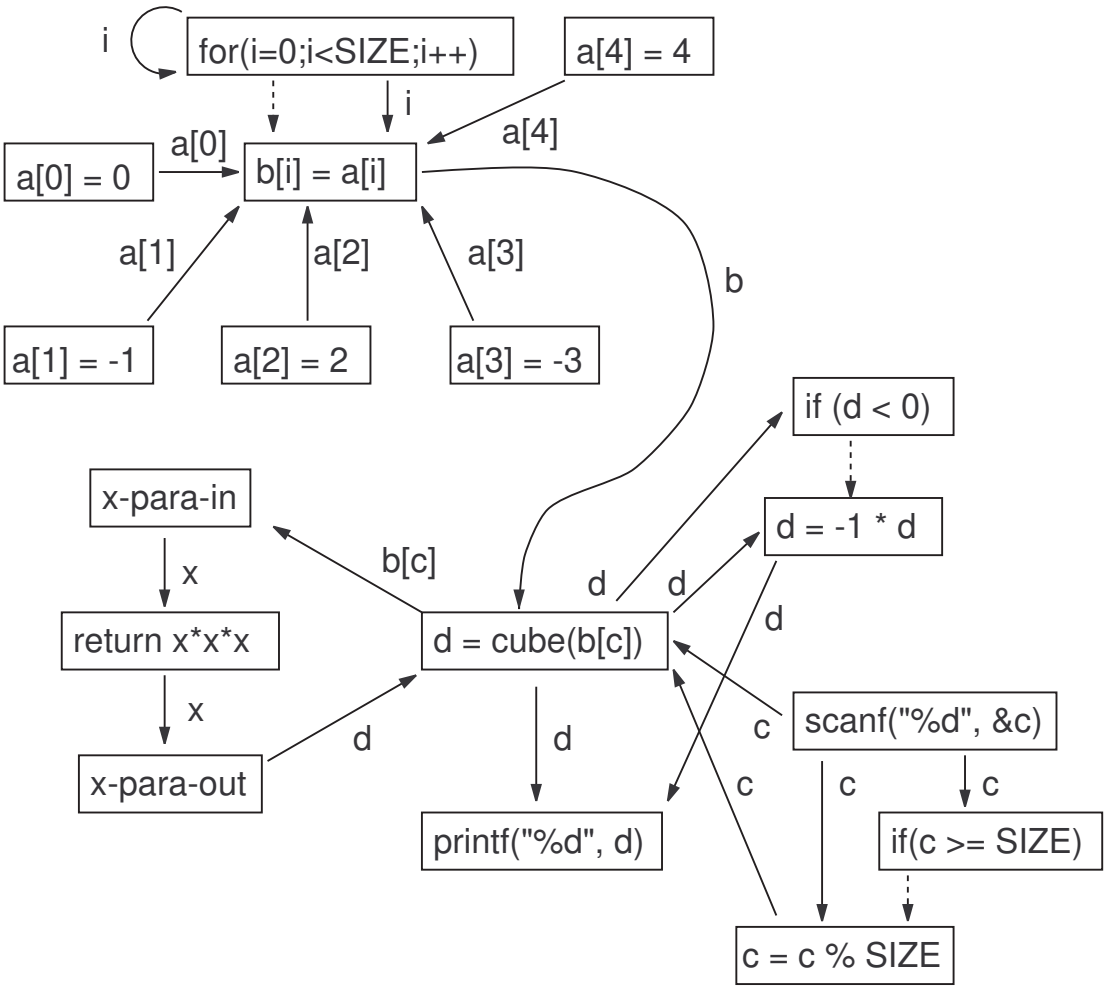


図 2: 図 1 のプログラムに対して静的に構築される PDG

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
31:      d = cube(b[c]);
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
37:      printf("%d\n", d);
38:  }
```

<37 , d>に関する静的スライス

図 3: 図 1 のプログラムの < 37 , $d >$ に関する静的スライス

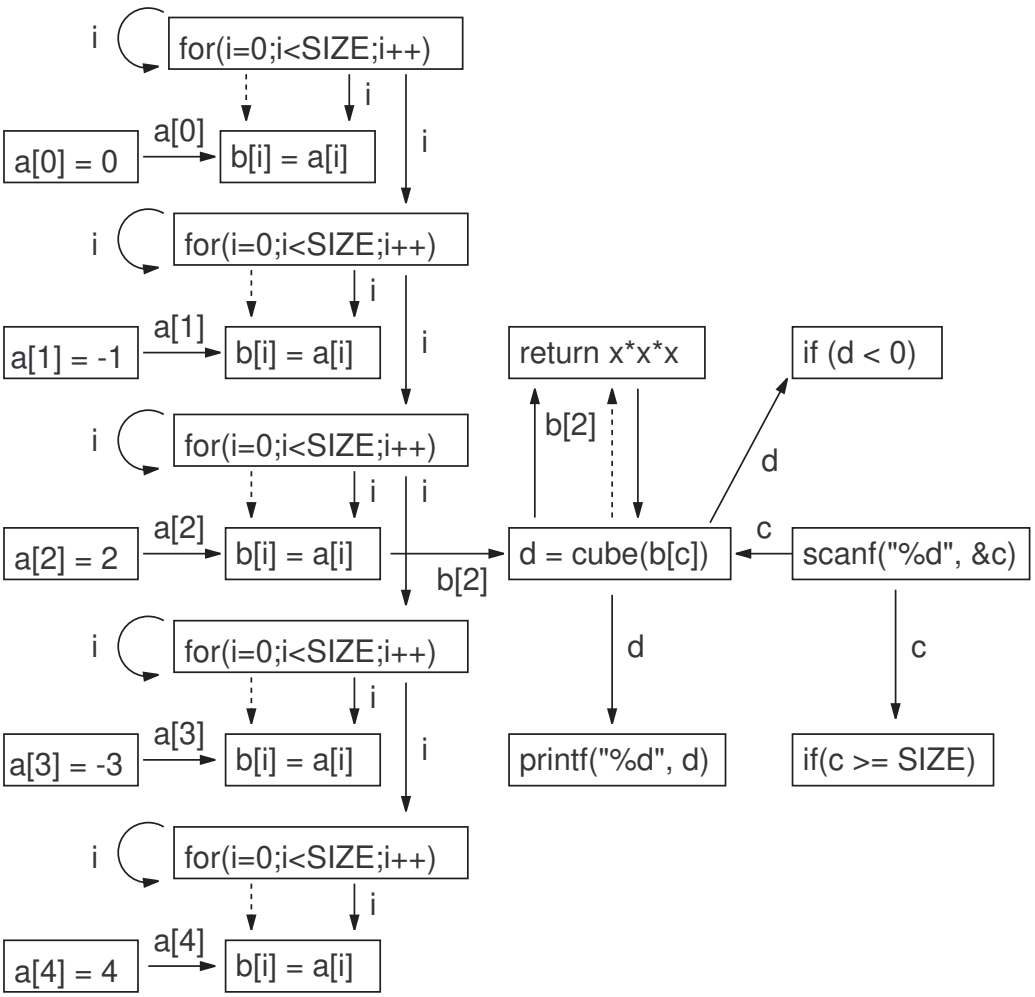


図 4: 図 1 のプログラムに対して動的に構築される PDG

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:  void main(void)
 4:  int a[SIZE];
 5:  int b[SIZE];
 6:  int c, d, i;
 7:  a[0] = 0;
 8:  a[1] = -1;
 9:  a[2] = 2;
10:  a[3] = -3;
11:  a[4] = 4;
12:  for (i=0; i<SIZE; i++)
13:  b[i] = a[i];
14:  for (i=0; i<SIZE; i++)
15:  b[i] = a[i];
16:  for (i=0; i<SIZE; i++)
17:  b[i] = a[i];
18:  for (i=0; i<SIZE; i++)
19:  b[i] = a[i];
20:  for (i=0; i<SIZE; i++)
21:  b[i] = a[i];
22:  scanf("%d", &c);
23:  if (c >= SIZE)
24:  d = cube(b[c]);
25:  int cube(int x)
26:  return x*x*x;
27:  if (d < 0)
28:  printf("%d\n", d);
```

図 5: 図 1 のプログラムに入力 2 を与えた場合の実行系列

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
16:      a[2] = 2;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
31:      d = cube(b[c]);
37:      printf("%d\n", d);
38:  }
```

<37 , d>に関する動的スライス

図 6: 図 1 のプログラムに入力 2 を与えた際の，$< 37 , d >$ に関する動的スライス

要に多くのデータ依存関係を生成してしまうことがある．また，ポインタを介したエイリアス（*Alias*）などによる陽に現れないデータ依存関係を考慮しなければならないため，静的なデータ依存関係解析には限界がある．

DC スライスは，スライス計算の Phase 1 における (a) 制御依存関係解析を静的に，(b) データ依存関係解析を動的に行うスライス計算手法である．動的にデータ依存関係解析を行うことにより，配列の添字やポインタの参照先などの実行時決定要素を正確に把握することができる．一方，制御依存関係解析については静的に行い，実行系列を保存する必要がなく，動的スライスに比べ解析コストを抑えることができる．

上述のように動的に抽出されるデータ依存関係と，静的に抽出される制御依存関係を用いて PDG を構築される．そして，他の手法と同様に，スライス基準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 スライスの例として，図 1 の C 言語で記述されたソースコードに対し，動的スライスと同様に入力 2 を与えて実行し，スライス基準 $< 37, d >$ に関する DC スライスを計算した結果を図 7 に，比較のため元のソースプログラムと併せて示す．

2.5 各スライス手法の比較

静的スライス，動的スライス，DC スライスの計算手法の違いを表 1 に示す．

表 1: 各スライス手法の違い

	静的スライス	DC スライス	動的スライス
CD 解析	静的	静的	動的
DD 解析	静的	動的	動的
PDG 節点	プログラム文	プログラム文	実行時点

また，計算手法の違いにより，解析精度（スライスサイズ），解析コスト（依存関係解析時間）に関し以下のような特性を持つ [4][15]．

解析精度（スライスサイズ）：静的スライス $\geq$ DC スライス $\geq$ 動的スライス

解析コスト（依存関係解析時間）：動的スライス $\gg$ DC スライス $>$ 静的スライス

これらのことから，DC スライスは，動的スライスより小さい解析コストで，静的スライスより高い精度のスライスを計算できる手法といえる．

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

サンプルコード

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
25:      scanf("%d", &c);
31:      d = cube(b[c]);
37:      printf("%d\n", d);
38:  }
```

<37 , d>に関する DC スライス

図 7: 図 1 のプログラムの < 37 , $d >$ に関する DC スライス

3　DC スライスの Java への適用

オブジェクト指向言語である Java において，データ格納の基本単位であるオブジェクト (または，クラスのインスタンス) は，データおよびそれを扱うメソッドの組み合わせにより構成される．そのため，Java プログラムに対してスライス計算を行う場合，実行メソッドを特定するためにインスタンスの型を把握しておかねばならない．

しかし，静的スライスではプログラム実行を伴わないため，実行時のインスタンスの型を静的に推測する必要がある．既存の手法 [14] を用いることで，いくつかの型に限定することは可能であるが，その型を一意に決定することは難しく，実行メソッドを特定するのは困難である．そのため，静的スライスは精度の面で限界があるといえる．

一方，動的スライスはプログラム実行を伴うため，実行時のインスタンスの型を容易に把握することが可能である．そのため，実行メソッドも特定できることから，精度の高いスライス計算が可能である．

3.1　概要

オブジェクト言語 Java は近年のソフトウェア開発環境で多く利用される．Java には多くの実行時決定要素が存在するので，静的スライスでは十分な解析精度が得られず，また動的スライスでは解析コストが膨大な量になってしまう．

そこで本研究では，Java を対象としたスライスシステムの構築として，DC スライスを適用する．また，バイトコードを解析の単位とすることで細粒度のスライスを計算し，その結果をソースコードに反映させる．

以降，DC スライスのバイトコードへの適用について述べる．

3.2　解析手順

DC スライスの計算は以下の手順により実現される．

DC スライス計算手順

Phase 1: ソースコード・バイトコード対応表の出力

ソースコードのトークン列とバイトコードとの対応関係を抽出する．

Phase 2: バイトコードに対する依存関係解析

各バイトコード命令に対し，

(a) 静的制御依存解析

(b) 動的データ依存解析

を行う．(b) については，文献 [9] において詳細が述べられているので，そちらを参照されたい．

Phase 3: PDG によるスライス計算

Phase 2 で求めた依存関係を利用し，バイトコードの各命令を節点とした PDG を構築し，ソースコード上で指定されたスライス基準をバイトコード上のスライス基準に変換し，それに対応する節点からグラフ探索を行うことでバイトコードでのスライスを計算する．その結果をソースコードに反映させる．

以降，各 Phase での処理について詳細を述べる．

3.3 Phase1: ソースコード・バイトコード対応表の出力

ソースコード・バイトコード間の変換は，Java コンパイラが出力する対応表を参照することにより行なう．

対応表に記述されている情報には以下のものがある．

- バイトコードのクラスファイル上での位置 (プログラムカウンタ , PC)
- バイトコード自身 (オペコード)
- ソースコードのソースファイル上での行番号
- ソースコードのソースファイル上での開始位置
- ソースコードのソースファイル上でのサイズ

今回機能拡張した Java コンパイラは，上記の情報を対応表に与える．

Java コンパイラは，ソースファイルから構文解析木を構築する時に各節点に対応するソースコードの行番号と開始位置を保持させている．そこで各節点にソースコードのサイズを追加することでソースコードの対応するトークン列の情報を保持することができる．バイト

コードに関する情報は，コンパイラが構文解析木からバイトコードに変換する時にその情報を追加する．
構文解析木からバイトコードに変換された後にバイトコード全体に対して最適化が行なわれるが，最適化によって削除されるバイトコードは保持しているソースコードの情報も同時に失われるので，ソースコードとバイトコードの対応関係の整合性が失われる．そのため今回はバイトコードの最適化は行わない．

Java コンパイラによって出力される対応表の詳細を図 8 に示す．

```
class sample {
    public static void main(String args[]) {
        int i = 0;
        if (i < 5) {
            i++;
        } else {
            i--;
        }
        int j = i;
    }
}
```

PC	オペコード		行	位置	size	tokens
242	3	(iconst_0)	3	69	1	”0”
243	60	(istore_1)	3	65	3	”i =”
244	27	(iload_1)	4	77	1	”i”
245	8	(iconst_5)	4	81	1	”5”
246	162	(if_icmpge)	4	79	1	”<”
249	132	(iinc)	5	91	3	”i++”
252	167	(goto)	4	73	2	”if”
255	0	(nop)	4	79	1	”<”
256	0	(nop)	4	73	2	”if”
257	132	(iinc)	7	111	3	”i−−”
260	0	(nop)	4	73	2	”if”
261	27	(iload_1)	9	128	1	”i”
262	61	(istore_2)	9	124	3	”j =”
263	177	(return)	2	38	4	”main”

図 8: Java コンパイラが出力するバイトコードとソースコードの対応表

一般的なソースコードにおけるバイトコードとの対応関係の例を以下に示す．

```
z = x + y - 1;
```

バイトコード	対応するトークン集合
iload_2	”x”
iload_3	”y”
iadd	”+”
iconst_1	”1”
isub	”−”
istore_1	”z =”

図 9: 代入文

```
if (i < 5) {
    i++;
}
```

バイトコード	対応するトークン集合
iload_1	”i”
iconst_5	”5”
if_icmplt L8	”<”
iinc 1 1	”i++”
L8: nop	”<”
nop	”if”

図 10: else 節のない if 文

```
if (i < 5) {
    i++;
} else {
    i--;
}
```

バイトコード	対応するトークン集合
iload_1	”i”
iconst_5	”5”
if_icmplt L13	”<”
iinc 1 1	”i++”
goto L18	”if”
L13: nop	”<”
nop	”if”
iinc 1 -1	”i−−”
L18: nop	”if”

図 11: else 節のある if 文

```
for(i=0;i<5;i++){
    j++;
}
```

バイトコード	対応するトークン集合
iconst_0	”0”
istore_1	”i =”
goto L14	”for”
nop	”for”
nop	”<”
iinc 2 1	”j++”
nop	”for”
iinc 1 1	”i++”
L14: nop	”for”
iload_1	”i”
iconst_5	”5”
if_icmplt L23	”<”
iinc 1 1	”i++”
L23: nop	”if”

図 12: for 文

```
while (i < 5) {
    i++;
}
```

バイトコード	対応するトークン集合
goto L8	”while”
nop	”while”
nop	”<”
iinc 1 1	”i++”
L8: nop	”while”
iload_1	”i”
iconst_5	”5”
if_icmplt L13	”<”
L13: nop	”while”

図 13: while 文

3.4 Phase2(a): 静的制御依存関係解析

Phase 2(a) では，与えられたバイトコードに対し，制御依存関係解析を静的に行う．その際，ソースコードにおける条件節とその述部という概念に基づく制御依存関係の定義を，そのままバイトコードに適用することは困難であるため，本研究では，バイトコードでの制御依存関係を [3] に従い定義する．以降，まずこれからの議論に必要な用語について説明し，バイトコードにおける制御依存関係の定義を行う．

制御フローグラフ

制御フローグラフ (*Control Flow Graph* , CFG) とは , プログラムの制御の流れをグラフで表したものである . 制御フローグラフは , プログラム中の分岐も合流もない部分 (基本ブロック[2] , *basic block*) を節点とし , それらの間を分岐や合流を表す有向辺で表した有向グラフである .

基本ブロックは , 文 (*statement*) の列で , その間には分岐も合流もない . 基本ブロックの先頭には一般に合流があり , 最後からは分岐がある . 基本ブロック中の文は先頭から最後まで一直線に実行される .

また , 制御フローグラフにおいて , 節点 X から節点 Y に向かって有向辺が引かれているとき , X は Y の先行ノード (*predecessor node*) , Y は X の後続ノード (*successor node*) という . X の先行ノードの集合を $Pred(X)$, 後続ノードの集合を $Succ(X)$ と表す .

上記の制御フローグラフは , 図 14 に示すアルゴリズムにより構築できる . なお , このアルゴリズムはメソッド単位に適用する .

図 15 に制御フローグラフの例を示す . この例ににおいて , $Pred(2) = \{1, 8\}$, $Succ(2) = \{3, 7\}$ である .

入力 メソッド単位のバイトコード
出力 バイトコードにおける制御フローグラフ
処理 バイトコードを基本ブロックに分割し，各基本ブロック間の制御の流れの関係を有向辺で表し，制御フローグラフを構築する

(1) /* first step */
(2) 最初の命令から，最初の分岐命令までを一つのブロックとして区切る
(3) **while** 次の命令が存在する **begin**
(4) 次の命令から次の分岐命令までを一つのブロックとして区切る
(5) 一つ前のブロックから制御が移行してくる可能性があれば前のブロックからこのブロックに有向辺を引く。
(6) **end**
(7) /* second step */
(8) **foreach** x（x は各ブロック B_1 の最後の命令） **begin**
(9) **foreach** B_2（B_2 は x のオペランドで指定された番地 a を含むブロック） **begin**
(10) **if** a がブロック B_2 の先頭である **then begin**
(11) ブロック B_1 からブロック B_2 に有向辺を引く
(12) **end**
(13) **else begin**
(14) ブロック B_2 を番地 a より前と以降とで二つのブロックに分割する
(15) 前半部のブロックから後半部のブロックに有向辺を引く
(16) ブロック B_1 から後半部のブロックに有向辺を引く
(17) **end**
(18) **end**
(19) **end**

図 14: バイトコードにおける制御フローグラフ構築アルゴリズム

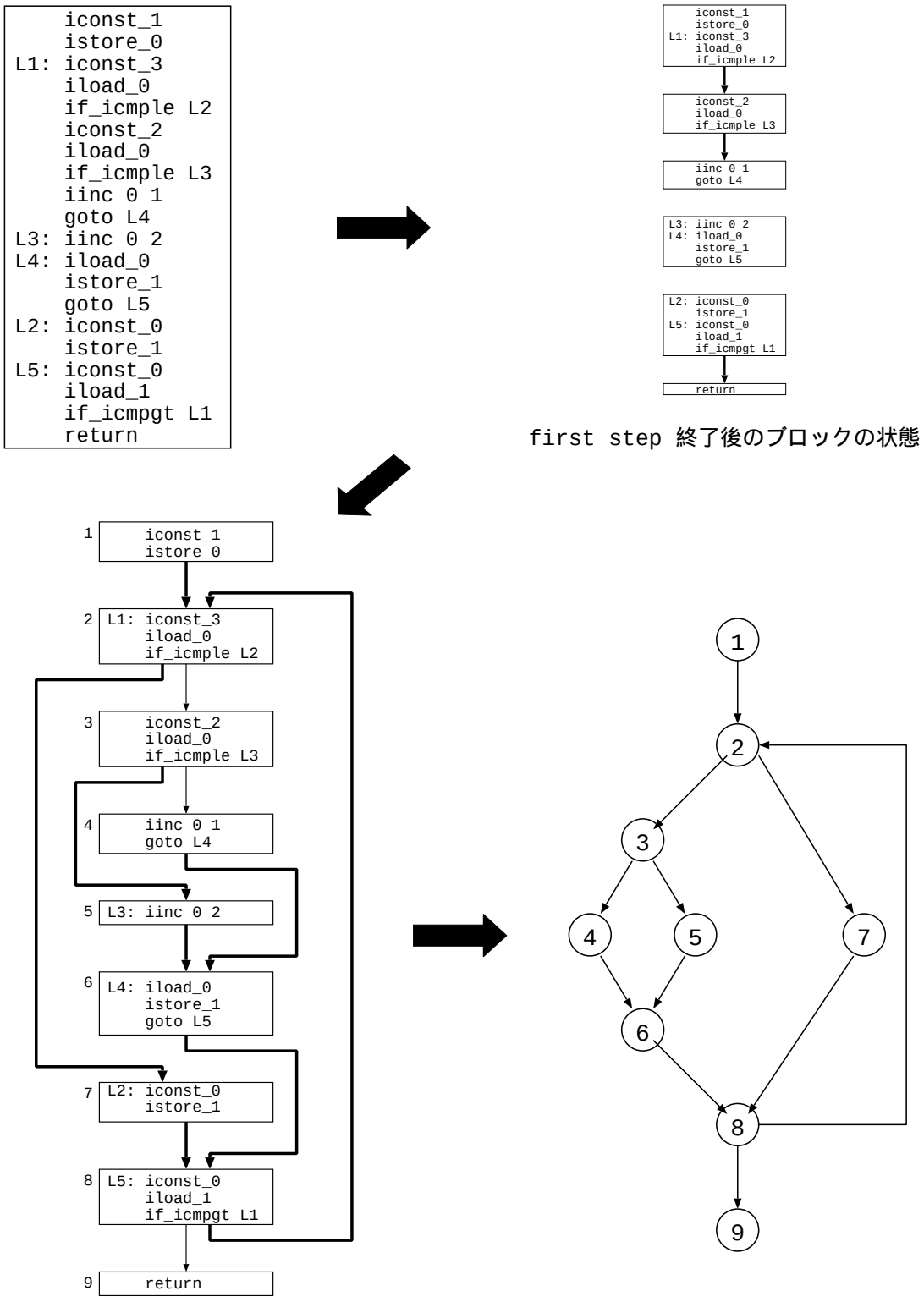


図 15: バイトコードに対して構築される制御フローグラフ

支配木

X，Y を制御フローグラフの節点としたとき，その入口節点から Y に至る全ての経路（$path$）に X が現れるとき，X は Y を支配する（$dominate$）といい $X \underline{\gg} Y$ と表す．支配関係は反射的（$reflexitive$）であり，また，推移的（$transitive$）であるため，X は自分自身を支配する．また，X が Y を支配し，Y が Z を支配するならば X は Z を支配する．

X が Y を支配し，かつ $X \neq Y$ のとき，X は Y を厳密に支配する（$strictly\ dominate$）といい $X \gg Y$ と表す．X が Y を厳密に支配し，X から Y への経路に X 以外に Y を厳密に支配する節点がないとき，X は Y を直接支配するといい $X = idom(Y)$ と表す．節点間の直接支配の関係を辺で表した木構造を支配木（$dominator\ tree$）という．支配木の根は入口節点となる．X の親（$parent$）を $parent(X)$，子の集合を $Children(X)$ と表す．

上記の支配木は，図 16，及び図 17 に示すアルゴリズムにより構築できる．また図 15 の制御フローグラフから構築される支配木を図 18 に支配木の例を示す．この例ににおいて，$parent(2) = 1$，$Children(2) = \{3, 7, 8\}$ である．

入力 制御フローグラフ
出力 制御フローグラフの節点 X を支配する節点の集合: $D(X)$
処理 制御フローグラフにおける支配節の検出

(1) /* 支配節を求める */

(2) $D(n_0) = \{n_0\}$ (N:制御フローグラフの節点集合, n_0:開始節)

(3) **foreach** $n \in N - \{n_0\}$ **do** $D(n) = N$;

(4) /* 初期設定終り */

(5) **do**

(6) **foreach** $n \in N - \{n_0\}$ **do**

(7) $D(n) = \{n\} \cup \bigcap D(p)$;(p は n の先行節)

(8) **until** $D(n)$ のどれにも変化がない

実行が終了した時点で $D(n)$ に含まれている節点が節点 n を支配する

図 16: 制御フローグラフにおける支配節検出アルゴリズム

入力 図 16 のアルゴリズムによって求められたすべての節点 X に対する $D(X)$
出力 制御フローグラフの節点 X が直接支配する節点の集合: $DT(X)$
処理 制御フローグラフにおける支配木を構築する

```
(1)  /* 支配節から支配木を求める */
(2)  /* 入口ノードから探索する */
(3)  call COMPUTE_DT(n_0)
(4)  /* 手続き COMPUTE_DT */
(5)  COMPUTE_DT(X):
(6)      DT(X) = ∅;
(7)      foreach {Y|X ∈ D(Y)} do
(8)          if Y が自分自身以外に支配しているノードがない then
(9)              DT(X) = DT(X) ∪ {Y};
```

図 17: 制御フローグラフにおける支配木構築アルゴリズム

- $D(1) = \{1\}$
- $D(2) = \{1, 2\}$
- $D(3) = \{1, 2, 3\}$
- $D(4) = \{1, 2, 3, 4\}$
- $D(5) = \{1, 2, 3, 5\}$
- $D(6) = \{1, 2, 3, 6\}$
- $D(7) = \{1, 2, 7\}$
- $D(8) = \{1, 2, 8\}$
- $D(9) = \{1, 2, 8, 9\}$

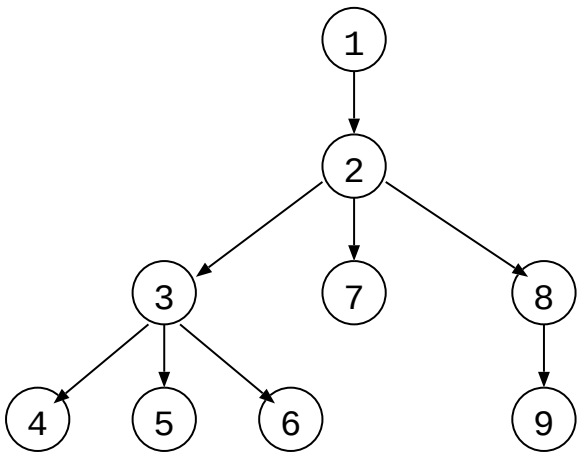


図 18: 図 15 の制御フローグラフから検出される支配節と構築される支配木

Dominance Frontier

制御フローグラフ上で，節点 X からグラフを辿り，初めて X の支配から外れた節点の集合を X の **Dominance Frontier** という．制御フローグラフの節点 X の Dominance Frontier とは，次の 2 つの条件を満たす節点 Y の集合である．

1. X は Y の先行ノード集合 $Pred(Y)$ の中の少なくとも 1 つの節点を支配する．
2. X は Y を厳密に支配しない．

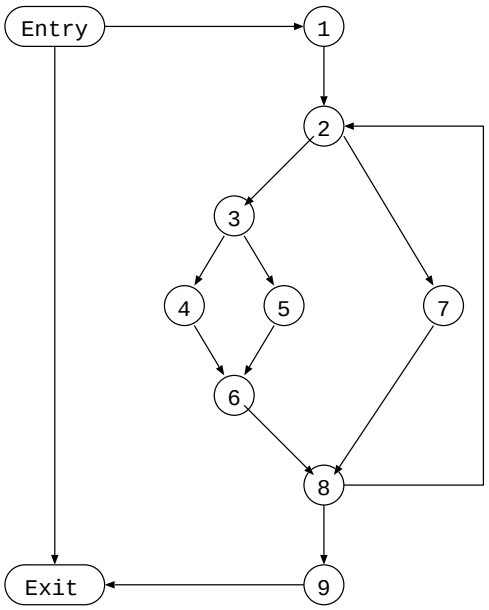
上記の Dominance Frontier は，図 19 に示すアルゴリズムにより検出できる．

支配木を構築するアルゴリズム [6][11] および Dominance Frontier を検出するアルゴリズム [5] は既に提案されている．また，図 15 の制御フローグラフに入口節点・出口節点を加え，その節点間に有向辺を引いた制御フローグラフに対して構築した支配木を図 20 に示す．

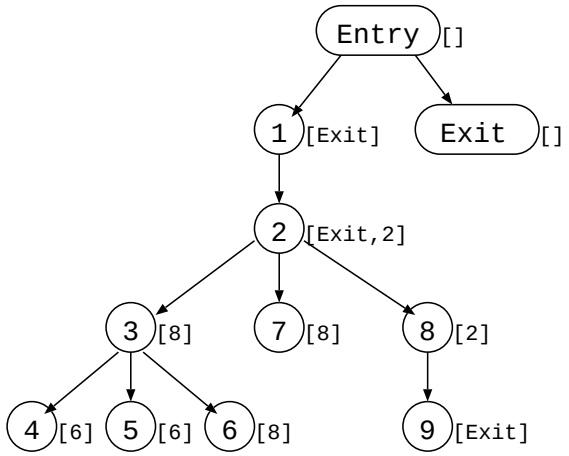
入力 入口節点と出口節点を加えた制御フローグラフ
出力 節点 X の Dominance Frontier の集合: $DF(X)$
処理 制御フローグラフにおける Dominance Frontier の検出

```
(1)  COMPUTE_DF(X):
(2)     DF(X) = ∅;
(3)     /* DF_local(X) を求める部分 */
(4)     foreach Y ∈ Succ(X) do
(5)        if idom(Y) ≠ X do
(6)           DF(X) ← DF(X) ∪ {Y}
(7)     foreach Z ∈ Children(X) do
(8)        call COMPUTE_DF(Z)
(9)        /* DF_up(Z) を求める部分 */
(10)       for each Y ∈ DF(Z) do
(11)          if idom(Y) ≠ X  then
(12)             DF(X) ← DF(X) ∪ {Y}
```

図 19: 制御フローグラフにおける Dominance Frontier 抽出アルゴリズム



(a):制御フローグラフ



(b):(a) に対する支配木 ([ ] 内は Dominance Frontier)

図 20: 制御フローグラフと支配木,Dominance Frontier

本研究では，バイトコードにおける制御依存関係を次のように定義する．

バイトコードにおける制御依存関係

バイトコードに対して構築された制御フローグラフの節点 X に属する命令 s，節点 Y に属する命令 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が存在するという．

1. s は節点 X の最終命令であり，分岐命令である．
2. 節点 X から U と V への分岐があり，U から出口へ Y を通らないパスがあり，V から出口へのパスが必ず Y を通る．

このように定義したバイトコードにおける制御依存関係は，図 21 に示すアルゴリズムにより抽出できる．なお．このアルゴリズムはメソッド単位に適用する．

入力 バイトコード
出力 命令間に存在する制御依存関係
処理 バイトコードにおける静的制御依存関係を抽出する

(1) バイトコードを基本ブロックに分割し，制御フローグラフ G を構築する

(2) G に入口節点 R，出口節点 E を追加し，R から G の最初の節点 S へ，R から E にそれぞれ辺を追加する

(3) G に対し，逆制御フローグラフ G' を構築する ($\mathcal{N}$: G' の節点集合)

(4) G' に対し，支配木を構築する (根は G の出口節点となる)

(5) **foreach** x **in** $\mathcal{N}$ **begin**

(6) Dominance Frontier $DF_G'[x]$ を計算する

(7) **foreach** y **in** $DF_G'[x]$

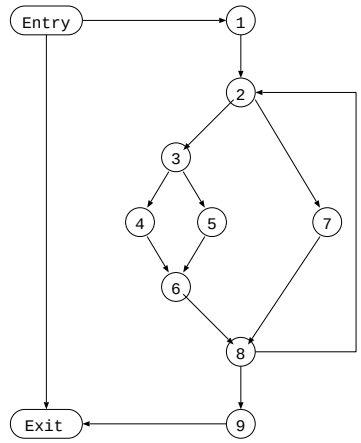
(8) x の最終命令と y の各命令の対を制御依存関係として抽出する

(9) **end**

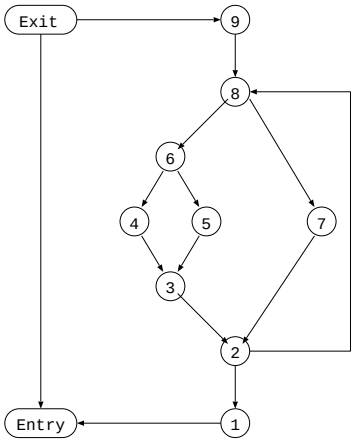
図 21: バイトコードにおける静的制御依存関係解析アルゴリズム

このアルゴリズムを適用することによって抽出される制御依存関係の例を図 22 に示す．

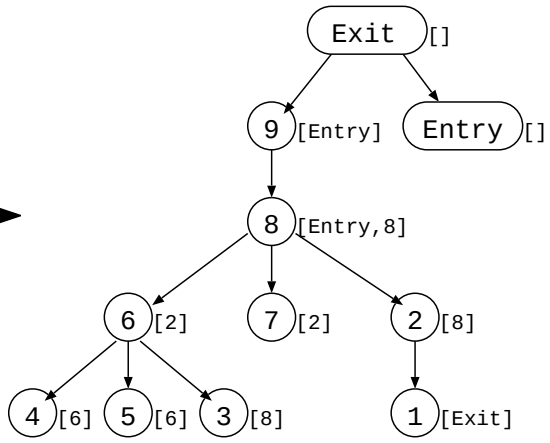
```
 1:      iconst_1          (1)
 2:      istore_0          (1)
 3: L1:  iconst_3          (2)
 4:      iload_0           (2)
 5:      if_icmple L2      (2)
 6:      iconst_2          (3)
 7:      iload_0           (3)
 8:      if_icmple L3      (3)
 9:      iinc 0 1          (4)
10:      goto L4           (4)
11: L3:  iinc 0 2          (5)
12: L4:  iload_0           (6)
13:      istore_1          (6)
14:      goto L5           (6)
15: L2:  iconst_0          (7)
16:      istore_1          (7)
17: L5:  iconst_0          (8)
18:      iload_1           (8)
19:      if_icmpgt L1      (8)
20:      return            (9)
```



制御フローグラフG



逆制御フローグラフG'



支配木（[ ]内はDominance Frontier）

制御依存関係のある命令の組	
(5: if_icmple L2 , 6: iconst_2)	(5: if_icmple L2 , 7: iload_0)
(5: if_icmple L2 , 8: if_icmple L3)	(5: if_icmple L2 , 12: iload_0)
(5: if_icmple L2 , 13: istore_1)	(5: if_icmple L2 , 14: goto L5)
(5: if_icmple L2 , 15: iconst_0)	(5: if_icmple L2 , 16: istore_1)
(5: if_icmple L2 , 16: iadd)	(5: if_icmple L2 , 17: istore_1)
(8: if_icmple L3 , 9: iinc 0 1)	(8: if_icmpgt L3 , 10: goto L4)
(8: if_icmple L3 , 11: iinc 0 2)	(19: if_icmpgt L1 , 3: iconst_3)
(19: if_icmpgt L1 , 4: iload_0)	(19: if_icmpgt L1 , 5: if_cmple L2)
(19: if_icmpgt L1 , 17: iconst_0)	(19: if_icmpgt L1 , 18: iload_1)
(19: if_icmpgt L1 , 19: if_cmpgt L1)	

図 22: バイトコードに対して抽出される制御依存関係

3.5 Phase3: PDG によるスライス計算

静的制御依存関係解析，動的データ依存関係解析により抽出された依存関係を用いてPDGを構築する．構築されるPDGの例を図23に示す．PDGの節点はバイトコードの各命令であるため，実行系列を保存する必要はなく，動的スライスに比べ解析コストは十分に小さくなる．

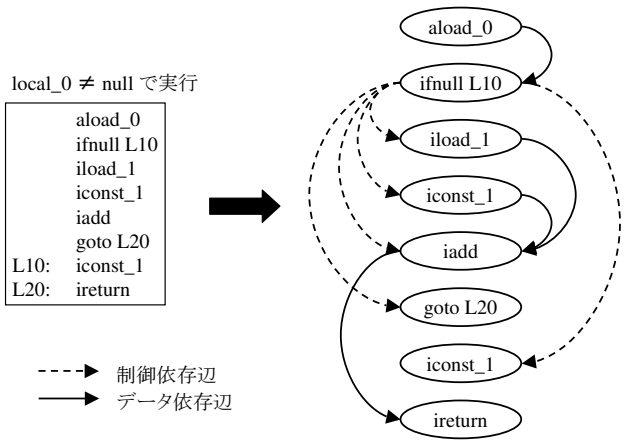


図 23: バイトコードにおけるプログラム依存グラフ

PDGを用いてスライス計算を行う．バイトコードに対するスライス計算も従来手法と同じく，スライス基準に対応するPDG節点からPDG辺を逆に辿り，到達可能な節点集合を求めることによる．

たとえば，図23のPDGにおいて、命令 ireturn に対応する節点をスライス基準としてバイトコードにおけるスライス計算を行うと，図24で斜線で示された節点が到達可能であり，バイトコードに対して計算されたスライス結果となる．

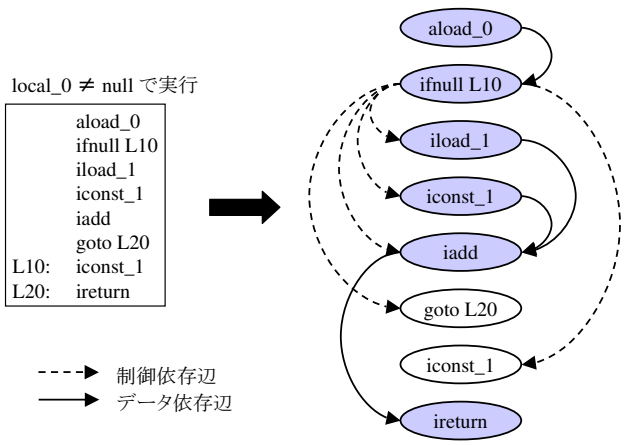


図 24: バイトコードにおける PDG によるスライス計算

ここで，図 25 のソースコードを実行させ，スライスを計算する過程を図 26 に示す．まず，バイトコードに対する静的制御依存関係解析・動的データ依存関係解析により PDG が構築される．次に，ソースコード上で指定されたスライス基準を，図 25 の対応表を参照することでバイトコードに変換し，対応する PDG 節点を求める．求められた PDG 節点からグラフ探索を行うことで，バイトコードにおけるスライスが計算される．最後に，計算されたバイトコードでのスライスを，再び対応表を参照することでソースコードに対応付ける．

```
int i = 0;
if (i < 5) {
    i++;
} else {
    i--;
}
int j = i;
```

バイトコード	対応するトークン集合
iconst_0	”0”
istore_1	”i =”
iload_1	”i”
iconst_2	”5”
if_icmpge L13	”<”
iinc 1 1	”i++”
goto L18	”if”
L13: nop	””
nop	”if”
iinc 1 -1	”i−−”
L18: nop	”if”
iload_1	”i”
istore_2	”j =”
return	”main”

図 25: ソースコード及び対応表

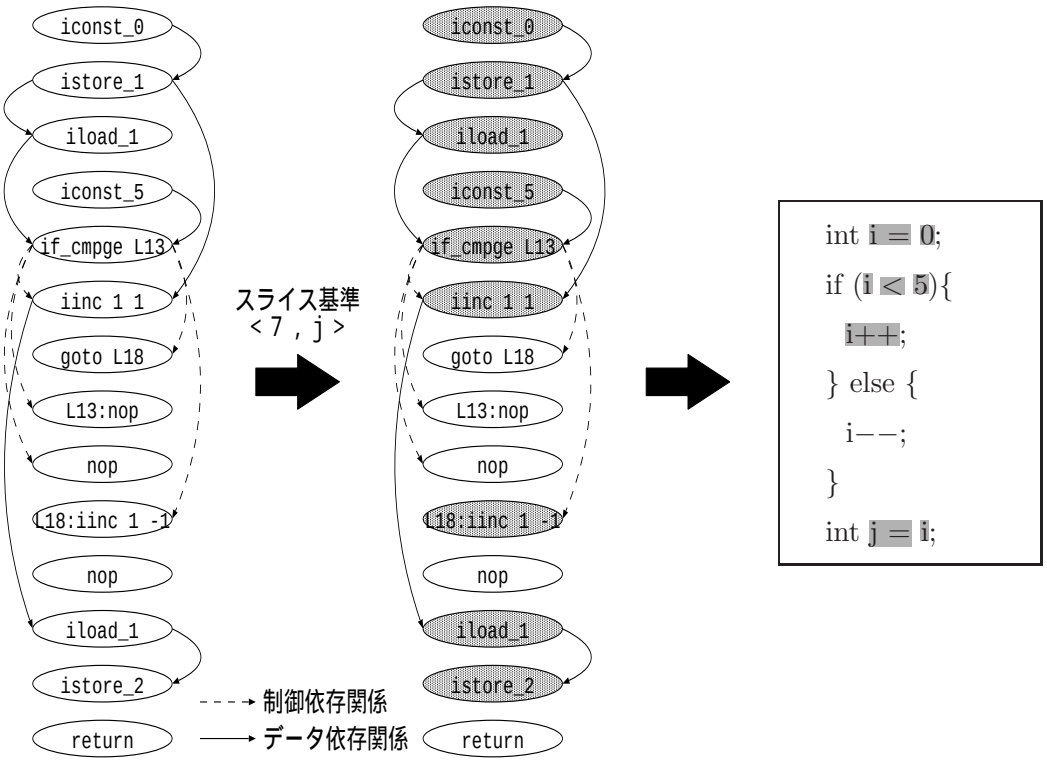


図 26: 構築された PDG からのスライスの抽出

4　提案手法の実現

4.1　システム構成

ソースコードのコンパイル時にバイトコードとソースコードの対応表を生成する．そして，静的に制御依存関係解析を行ったのち，JVM 上でバイトコードの実行を行いながら動的にデータ依存関係解析を行う．これらにより抽出された依存関係を元にバイトコードの各命令が節点となる PDG を構築する．ユーザによりソースコードにおけるスライス基準が指定されると，対応表を用いてそれをバイトコードにおけるスライス基準に変換し，PDG 探索よるスライス計算を行う．最後に，対応表を参照しながらスライス結果をソースコードに対応付ける．以降，Java コンパイラ，制御依存関係解析部，スライスエンジン部の実装について順に説明する．

Java コンパイラ

入力：　Java ソースコード
出力：　バイトコード・ソースコードのトークン集合とバイトコードの対応表
処理：　入力ファイルを解析し，通常のコンパイラと同様にコンパイルし，クラスファイルを出力する．また，生成されるバイトコードの各命令と，それに対応するソースコードのトークン集合の対応表を出力する．
概要：　JDK 付属の `javac` に対する機能拡張として実装．
実装言語：　Java
クラス数：　174 (149)
行数：　約 42,000 行 (約 3,000 行)
　　( ) 内は今回の機能拡張によって追加・修正されたコードを表す．

制御依存関係解析部

入力：　バイトコード
出力：　静的に解析することによって抽出できるバイトコードの命令間の制御依存関係
処理：　入力されたバイトコードに対し，静的に制御依存関係解析を行い，各命令間に存在するデータ依存関係を静的に抽出し，その結果を出力する．
概要：　バイトコードダンプツール `jaca`[8] に対する機能拡張として実装．
実装言語：　C
ファイル数：　30 (19)
行数：　約 4,000 行 (約 1,600 行)
　　( ) 内は今回の機能拡張によって追加・修正されたコードを表す．

スライスエンジン部

入力： 依存関係解析の結果・ソースコードのトークン集合とバイトコードの対応表

出力： ソースコードにおけるスライス

処理： 入力された依存関係解析の結果からバイトコードでの PDG を構築する．また，ユーザから指定されたスライス基準を，対応表を用いることでバイトコードの命令に変換し，バイトコードにおけるスライスを計算する．計算されたスライスを，再び対応表を用いることでソースコードにおけるスライスに変換する．

実装言語： C

ファイル数： 15

行数： 約 1,200 行

4.2 評価

前節で述べたシステムを用いて提案手法の有効性を評価した．ここでは，以下の点について比較を行う．

1. 計算されるスライスサイズ
2. 解析コスト (解析時間・解析時使用メモリ量)

スライスサイズについては，静的スライス，動的スライスとの比較を行った．スライス計算を行うシステムの実装は，提案手法のみであるため，提案手法のスライスは実装システムを用いて求め，他の手法のスライスは手計算で求めた．

解析コストは，各処理にかかる時間・使用メモリ量を測定することにより評価を行った．コンパイルについては通常のコンパイルとの比較を行い，制御依存関係解析・PDG 構築およびスライス計算については各処理に必要な時間と使用メモリ量を測定することで評価を行った．

スライス対象プログラムの概要を表 2 に示す．P1 は，ファイルからデータを読み込み，そのデータを操作し，再び保存することの出来る簡易データベースシステムである．また，P2 は 50 個の整数値に対し，バブルソート・双方向バブルソート・クイックソートを適用するプログラムである．

表 2: スライス対象プログラム

プログラム	クラス数	オーバーライドメソッド数	総行数
P1	4	0	262
P2	5	3	231

4.2.1 スライスサイズ

Java を対象とした静的スライス，動的スライスによって得られたスライスと提案手法によって得られたスライスのサイズの比較を行った．各手法ともに，計算の結果スライスに含まれるべき文含まれないということはないため，得られたスライスのサイズが小さいほど精度が高い (スライスに含まれるべきではない文がスライスに含まれる割合が小さい) といえる．

計測値は，各プログラムに対し，任意に定めた 2 つのスライス基準について，それぞれの手法で得られたスライスのサイズである．実装は，本手法のみ行ったため，提案手法のスライスは実装したシステムを用いて求め，他の手法のスライスは手計算で求めた．計測値を表 3 に示す．

表 3: スライスサイズ [行] (全体に対するスライスの割合)

	静的スライス	動的スライス	提案手法
P1-スライス基準 1	60 (22.9%)	24 (10.3%)	30 (11.4%)
P1-スライス基準 2	19 (7.3%)	14 (5.4%)	15 (5.7%)
P2-スライス基準 1	79 (34.2%)	51 (22.1%)	51 (22.1%)
P2-スライス基準 2	27 (11.7%)	23 (10.0%)	25 (10.8%)

提案手法で得られるスライスは，静的スライスの約 50%から 93%のサイズであり，静的スライスより高精度なスライスを得られることがわかった．今回の実験は，スライス対象プログラムが比較的小規模であったため，提案手法によるスライスと静的スライスの差は少なかったと考えられる．

しかし，クラスの継承やメソッドのオーバーライド，オーバーロードがより多く含まれる大規模プログラムでは，静的スライスの精度がより低下することが推測できる．それに対し，提案手法では，継承やオーバーライド，オーバーロードの数に関わらず，それによる精度の

低下は発生しない．また，今回の実験においては，提案手法では動的スライスとほぼ同等のスライスが得られた．

4.2.2 解析コスト

各処理について，処理時間および使用メモリ量を計測することでスライス計算に必要なコストの評価を行った．実行環境を表 4 に示す．

表 4: 実験環境

CPU	Pentium 4 1.5GHz
メモリ	512MB
OS	FreeBSD 5.0-CURRENT
Java	JDK 1.2.2

以降，それぞれの処理について，計測したコスト値を順に示す．また，各計測値は，実行を 10 回行った際の平均値である．

Java コンパイラ

処理時間・使用メモリ量の評価を，通常のコンパイラとの比較により行った．測定結果を表 5・表 6 に示す．

表 5: コンパイル時間

プログラム	通常 [ms]	対応表出力 [ms]	対応表出力/通常
P1	1,114	2,405	2.16
P2	974	1,325	1.36

表 6: コンパイル時の使用メモリ量

プログラム	通常 [Kbytes]	対応表出力 [Kbytes]	対応表出力/通常
P1	11,490	11,824	1.03
P2	10,394	11,792	1.14

制御依存関係解析

処理時間・使用メモリ量の評価を，表 2 に示したスライス対象プログラムおよび約 4,800

個のクラスからなる JDK ライブラリを解析した際の解析時間と使用メモリ量の計測によって行った．計測値を表 7・表 8 に示す．

表 7: 静的制御依存関係解析時間

プログラム	総クラス数	解析時間（全体） [ms]	平均解析時間（一クラス） [ms]
P1	4	25.79	6.45
P2	5	24.56	4.92
JDK ライブラリ	4,807	48,060	9.99

表 8: 静的制御依存関係解析時の使用メモリ量

クラス名	サイズ [bytes]	使用メモリ量 [Kbytes]
java.security.Principal.class	264	124
sun.io.CharToByteCp866.class	7,269	181
sun.io.CharToByteCp933.class	193,709	2,779

PDG 構築・スライス計算

処理時間・使用メモリ量の評価を，表 2 に示したスライス対象プログラムに対し，あるスライス基準からのスライス計算時の PDG 構築・スライス計算 (PDG 探索) 時間の計測によって行った．計測結果を表 9・表 10 に示す．

表 9: PDG 構築・スライス計算時間

プログラム	PDG 節点数	PDG 構築時間 [ms]	スライス計算時間 [ms]	合計時間 [ms]
P1	34,966	346	179	525
P2	34,956	352	98	450

表 10: PDG 構築・スライス計算時の使用メモリ量

プログラム	PDG 節点数	使用メモリ量 [Kbytes]
P1	34,966	5,426
P2	34,956	4,047

PDG 節点数の評価

動的スライスを計算する場合のコストとの比較を行うために，表 2 に示したスライス対象プログラムに対して，提案手法によって構築された PDG の節点数と，動的スライス計算の際に保存される実行系列に含まれる命令数を計測した．計測結果を表 11 に示す．

表 11: 構築される PDG の節点数と解析したクラス数

プログラム	提案手法	動的スライス	提案手法/動的スライス	解析したクラス数
P1	34,966	1,198,596	0.0292	147
P2	34,956	1,808,051	0.0193	148

以上より，本システムは全く解析を行わずに通常実行を行う場合と比較し，多くの解析コストを要するといえる．これは動的データ依存関係の解析コストに加えて，コンパイラにおいてソースコードの対応関係を容易に取得できるように，バイトコードの最適化を一切行っていないことが原因である．

しかし，動的スライスの計算には，節点数の保存のためにおよそ 30 倍から 50 倍のコストが必要となり，4.2.1. で行った実験においてほぼ同等のスライス結果が得られたことを考慮すると，提案手法を用いることで動的スライスより格段に小さな解析コストで，静的スライスより高い精度のスライスの計算が可能となるといえる．

提案手法の今後の課題として，解析速度の点について改良を行い，より低コストでスライスの計算を行うことを挙げることができる．しかし現段階で実用的に動作するスライスシステムは本システム以外にはなく，その点においても提案手法を実現した本システムは非常に有用であるといえる．

5　むすび

本研究では，オブジェクト指向言語 Java で記述されたプログラムを対象とするスライスシステムの試作を行い，その有効性を検証した．DC スライスの採用することで高精度のスライスを低コストで抽出することが可能となり．依存関係解析をバイトコード単位で行うことによる解析精度の向上も得られた．

試作システムは，ソースコード・バイトコード対応表出力を行う Java コンパイラ，バイトコード間の動的データ依存関係解析を行う JVM，バイトコードを対象とする静的制御依存関係解析ツール，PDG に基づくスライサで構成されている．また，実利用を考慮した実装となっており，JDK クラスライブラリのような大規模プログラムの解析も容易である．

今後の課題としては，以下が挙げられる．

- バイトコードの最適化に対応したバイトコード・ソースコード対応表の作成
- 例外処理などにより実行時に決定される，制御依存関係に対する動的解析の適用
- 実際のプログラム開発におけるシステムの適用

謝辞

本研究の全過程を通して，常に適切な御指導および御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 井上 克郎 教授に心より深く感謝致します．

本論文を作成するにあたり，逐次適切な御指導および御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 楠本 真二 助教授に心から感謝致します．

本論文を作成するにあたり，適切な御指導，御助言を賜りました 大阪大学大学院 基礎工学研究科 情報数理系専攻 松下 誠 助手に心から感謝致します．

本研究を通して，適切な御助言を頂きました 大阪大学大学院 基礎工学研究科 情報数理系専攻 大畑 文明 氏に深く感謝致します．

本研究を通して，適切な御助言を頂きました 大阪大学大学院 基礎工学研究科 情報数理系専攻 誉田 謙二 氏に深く感謝致します．

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 基礎工学研究科 情報数英系専攻 井上研究室の皆様に深く感謝いたします．

参考文献

[1] H.Agrawal and J.Horgan: “Dynamic Program Slicing”, SIGPLAN Notices, Vol.25, No.6, pp.246–256 (1990).

[2] A.V.Aho , R.Sethi , J.D.Ullman: ”Compilers Principles,Techniques, and Tools”, Addison-Wesley Publishing Company(1986) .

[3] A.W.Appel and M.Ginsburg: “Modern Compiler Implementation in C”, Cambridge University Press, Cambridge (1998).

[4] Y.Ashida, F.Ohata and K.Inoue: “Slicing Methods Using Static and Dynamic Information”, Proceedings of the 6th Asia Pacific Software Engineering Conference , pp.344–350, Takamatsu, Japan, December (1999).

[5] R.Cytron, J.Ferrante, B.K.Rosen, M.N.Wegman and F.K.Zadeck: “Efficiently Computing Static Single Assignment Form and the Control Dependench Graph”, ACM Transactions on Programming Languages and Systems, Vol.13, No.4, pp.461–486, October (1991).

[6] D.Harel: “A liner time algorithm for finding dominator in flow graphs and related problems”, Proceedings of 17th ACM Symposium on Theory of computing, pp.185–194, May (1985).

[7] K.Inoue, F.Ohata and Y.Ashida: “Lightweight Semi-Dynamic Methods for Efficient and Effective Program Slicing”, Technical Report of Osaka University, Department of Information and Computer Scineces, Inoue Laboratory, June (2000).

[8] E.Kawai: jaca, “http://minatow3.aist-nara.ac.jp/oie-lab/person/eiji-ka/research/my_jvm/jaca/”.

[9] 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, February (2002).

[10] L.Larsen and M.J.Harrold: “Slicing Object-Oriented Software”, Proceedings of the 18th International Conference on Software Engineering, pp.495–505, Berlin, March (1996).

[11] T.Lengauer and E.Tarjan: “A fast algorithm for finding dominators in a flow graph”, ACM Transactions on Programming Languages and Systems, Vol.1, No.1, pp.121–141, July (1979).

[12] F.Ohata, K.Hirose and K.Inoue: “A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information”, Proceedings of 8th Asia-Pacific Software Engineering Conference , pp.273–280, Macau, China, December (2001).

[13] K.J.Ottenstein and L.M.Ottenstein: “The program dependence graph in a software development environment”, Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

[14] B.Steensgaard: “Points-to analysis in almost linear time”, Technical Report MSR-TR-95-08, Microsoft Research (1995).

[15] 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, Vol.J85-D-I, No.2, pp.228–235, February (2002).

[16] R.Ueda, K.Inoue and H.Iida: “A Practical Slice Algorithm for Recursive Programs”, Proceedings of the International Symposium on Software Engineering for the Next Generation, pp.96–106, Nagoya, Japan, February (1996).

[17] M.Weiser: “Program Slicing”, Proceedings of the 5th International Conference on Software Engineering, pp. 439–449 (1981).

[18] J.Zhao: “Dynamic Slicing of Object-Oriented Programs”, Technical Report SE-98-119, Information Processing Society of Japan , pp.11–23, May (1998).