修士学位論文

題目

静的情報と動的情報を用いた Java プログラムスライス計算法

指導教官
井上克郎 教授

報告者
廣瀬航也

平成 13 年 2 月 14 日

大阪大学 大学院基礎工学研究科
情報数理系専攻 ソフトウェア科学分野

平成 12 年度 修士学位論文

静的情報と動的情報を用いた Java プログラムスライス計算法

廣瀬航也

内容梗概

プログラムデバッグを効率よく行うための手法として，プログラムスライスがある．一般的に，プログラムスライスはプログラム文間の制御依存関係およびデータ依存関係を解析することで得られる．静的スライスは制御依存関係解析，データ依存関係解析を共に静的に行うことで得られる．この手法は，低コストではあるが得られるスライスの正確性は低い．動的スライスは，制御依存関係解析，データ依存関係解析を共に静的に行うことで得られる．この手法は得られるスライスの正確性は高いが，多大なコストを要する．

現在のプログラム開発環境において，オブジェクト指向言語の利用が高まっている．オブジェクト指向言語には，従来の手続き型言語にはないクラス・継承・動的束縛などの新しい概念が導入されており，それらに対応した静的スライス計算法，動的スライス計算法が提案されている．

しかし，これらの手法は手続き型言語を対象としたスライス計算法の単純な拡張であり，それぞれ，正確性，コストの問題を解決できていない．

本研究では，オブジェクト指向言語 Java を対象としたプログラムスライス手法を提案する．Java は，オブジェクトへの参照・動的束縛など，実行時に決定される要素を多数含むため，静的解析では得られるスライスの正確性に限界がある．提案手法では，静的解析と動的解析を組み合わせることにより，スライスの正確性を高め，動的スライス計算法よりコストを抑えた．また，本手法をスライスツールとして実現し，その有効性を確認した．

主な用語

プログラムスライス (Program Slice)
静的解析 (Static Analysis)
動的解析 (Dynamic Analysis)
Java

目 次

1 まえがき **4**

2 プログラムスライス **6**

2.1 静的スライス 6

2.1.1 静的スライスを計算するアルゴリズム 6

2.1.2 静的スライスの適用例 7

2.2 動的スライス 11

2.2.1 動的スライスを計算するアルゴリズム 11

2.2.2 動的スライスの適用例 12

3 Dependence-Cache スライス **16**

3.1 計算手順 16

3.2 適用例 17

3.2.1 配列を含むプログラム 17

3.2.2 ポインタを利用するプログラム 17

3.3 他のスライス手法との比較 18

4 オブジェクト指向言語を対象としたスライス **21**

4.1 静的スライスの適用と問題点 21

4.2 動的スライスの適用と問題点 22

5 静的情報と動的情報を用いた Java スライス手法 **23**

5.1 提案手法 23

5.1.1 方針 23

5.1.2 アルゴリズム 24

5.1.3 動的データ依存関係解析アルゴリズム 24

5.2 適用例 24

6 本手法の実装 **30**

6.1 実装概要 30

6.2 キャッシュの実装 31

6.3 評価 32

6.3.1 スライスサイズ 33

6.3.2 解析コスト 34

6.4 考察 34

7　まとめと今後の課題 37

謝辞 38

参考文献 39

1 まえがき

現在の一般的なデバッグ作業においてはプログラム全体を扱うのが原則である．しかし，近年プログラムはより大規模で複雑なものになっている．このように大規模化，複雑化したプログラムではデバッグ時にエラー原因の特定に時間がかかってしまい，結果としてプログラム開発の効率が悪くなる．

このような場合に，エラーに関係がある可能性の高い部分だけをプログラムから抽出するような機能があれば，開発者はその部分だけに注目することができる．その結果，プログラム中のエラーの位置を発見する負担を軽減することができ開発効率の向上が期待される．

プログラムスライス($Program\ Slice$) は Weiser [15] によって提案されたものである．プログラム P のスライスとは，直感的には P 中のある地点 n のある変数 v に関して，n における v の値に影響を与え得る文や式の集合，つまり v に直接，または間接的に依存関係のある文や式の集合を言う．

このプログラムスライスを利用することにより，開発者がデバッグ時にエラーに関係のある部分のみに注目することができる．その結果，プログラム中のエラーの位置を発見する負担を軽減することができ，開発効率の向上が期待される．

当初，Weiser によって考案されたこのスライス抽出技法は，ソースプログラムのみを用いて依存関係を解析していた．それゆえ，この手法によって抽出されたスライスは静的スライス($Static\ Slice$) と呼ばれる．この手法では，スライスに含まれるべき文が排除されることを避けるために，全ての可能な入力データを考慮することになる．このため，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) も含んでしまうという欠点がある．

これに対して Agrawal [1] は，よりデバッグに有効な手法として動的スライス($Dynamic\ Slice$) を提案している．動的スライスの計算には，ソースプログラムの代わりに実行時に実際に実行された文，式の列 (実行系列) が用いられる．この手法では，特定の入力データのみを考慮することになり，その結果静的スライスと比較して，スライスサイズの減少 (正確性の向上) が期待できる．

デバッグに利用することを考えると，動的スライスは静的スライスと比較してより望ましい結果だということができる．しかし，実行系列はソースプログラムと比較して膨大な量になってしまう場合が多く，その保存と解析に多くの空間，時間コストを必要としてしまう．

スライスの計算に必要なコストとスライスサイズ (正確性) はトレードオフの関係にあり，さまざまな研究がなされている．その一つに，**Dependence-Cache(DC)** スライス($Dependence\text{-}Cache\ Slice$)[3] が挙げられる．DC スライス計算法は，配列やポインタを詳しく解析するためにデータ依存関係解析を動的に，必要コスト削減のために制御依存関係は静的に行う手法である．

現在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, C++等いわゆるオブジェクト指向言語の利用が高まっている．オブジェクト指向言語には，従来の手続き型言語にはないクラス，継承などの新しい概念が導入されており，既存のプログラムスライス計算手法をそのまま用いても正確なスライスが得られない．そこで，手続き型言語を対象としたスライス計算法のオブ

ジェクト指向言語への対応を試みた静的スライス [9]，動的スライス [16] が提案されている．

しかし，オブジェクト指向言語には，実行時に決定される要素が多く含まれるため，静的スライス計算法では得られるスライスの正確性に限界がある．一方，動的スライス計算法は手続き型言語を対象とした動的スライス計算法と同様に実行系列を保存するため，多大な時間，空間コストが必要となる．

本研究では，オブジェクト指向言語である Java を対象に，静的スライスと動的スライスの中間に位置する DC スライスを適用する手法を提案する．本手法では，DC スライス計算法と同様に動的にデータ依存解析を行うことで，実行時に決定されるデータ依存関係を正確に解析できる．また，Java はメソッドのオーバーロード，オーバーライド機能を持つため，静的な解析では呼び出されるメソッドを一意に決定することは困難である．そのため，スライスの正確性が大きく低下することが考えられる．そこで，メソッド間に存在する制御依存関係解析も動的に行うことで，呼び出されるメソッドを一意に決定し，スライスの正確性を高める．メソッド間以外の制御依存関係解析は静的に行い，また，実行系列の保存を行わないため，解析コストが動的スライスより小さくなる．

以降，2 ではプログラムスライスについて述べ，3 では，DC スライスについて述べる．4 では，既存のオブジェクト指向言語を対象としたスライスについて述べる．5 では，Java スライス手法について述べ，6 では，提案手法の実装と評価について述べる．最後に 7 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．スライスは，デバッグ，保守，プログラム理解等に利用される．

スライスのうち，全ての可能な入力データに関するスライスを静的スライスと呼び，ある特定の入力のもとで生成される実行系列を解析して求められたスライスを動的スライスと呼ぶ．

2.1 静的スライス

2.1.1 静的スライスを計算するアルゴリズム

静的スライスの計算方法として [10] のアルゴリズムを用いる．このアルゴリズムは [7, 14] を参考にしてプログラム中の文の依存関係を調べ，それを元にプログラム依存グラフ($Program\ Dependence\ Graph,\ PDG$) を作成し，その辺をたどることにより静的スライスを計算する．

- 諸定義

PDG はプログラム内の文の依存関係を表すグラフである．PDG の節点はプログラム中の各文及び if 文や while 文の条件判定部分を表し，辺は変数の影響を伝えるデータ依存関係($Data\ \ Dependence$) 及び条件文や繰り返し文の制御の影響を伝える制御依存関係($Control\ Dependence$) を表す．

データ依存関係は，各頂点の到達定義集合 (Reaching Definitions，略して RD) を求めることによって得られる．PDG 上でのある頂点 t の RD とは，変数 v と頂点 s との組 $\langle v, s\rangle$ の集合である．これは，

1. プログラム中の文 s で変数 v を定義している．
2. プログラム中の二つの文 s と t の間に v を必ず定義するような文がない．

ことを示している．t の RD に $\langle v, s\rangle$ が含まれ，かつ t が v を参照する時，s から t へのデータ依存関係 ($\mathrm{DD}(s, v, t)$) があるという．

また，ある条件判定部分 s の結果により文 t の実行の有無が決定される時，s と t との間に制御依存関係 ($\mathrm{CD}(s, t)$) が存在するものとする．すなわち制御依存関係は if 文や while 文の条件判定部分からそれらの内部ブロックに属する文への影響であり，これはプログラムを解析すれば容易に求められる．

一般にプログラムには複数の手続きが定義されている．各手続き間には引数や大域変数を通じてデータ依存関係が生じる．これらのデータ依存関係を表すために，PDG にプログラム中の文とは直接対応しない節点 (特殊節点と呼ぶ) を用意する．

PDG の作成は，プログラムを解析し，プログラムの各文を PDG の節点に切りわけ，プログラム中の各文における RD を求め，それをもとにして PDG の各辺を生成することによって行なわれる．詳細は，[14] を参照されたい．

上記の方法で生成された PDG を G，G に含まれる全ての辺の集合を E とすると，プログラム内の文 S の変数 v に関するスライスを表す節点の集合 V は以下のようにして計算される．

1. $V \leftarrow \{n | n$ は S に対応する節点 .$\}$
2. $N \leftarrow \{n | \langle v, n \rangle \in RD_{in}(S)\}$
3. $N \neq \phi$の間以下の動作を繰り返す．
   - $l \in N$ を一つ選ぶ．
   - $N \leftarrow N - \{l\}$
   - $V \leftarrow V \cup \{l\}$
   - $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l) \wedge (k$ が $g\text{–}in$ でない$) \wedge (k$ が $para\text{–}in$ 節点でない$)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す．)
   - k が $para\text{–}in$ 節点でも $para\text{–}in$ 節点でもないなら
     $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す．)

2.1.2 静的スライスの適用例

図 1 の C 言語で記述されたソースプログラムに対応する PDG は 図 2 のようになる．

図 1 のプログラムの，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．

```
 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: ソースプログラム

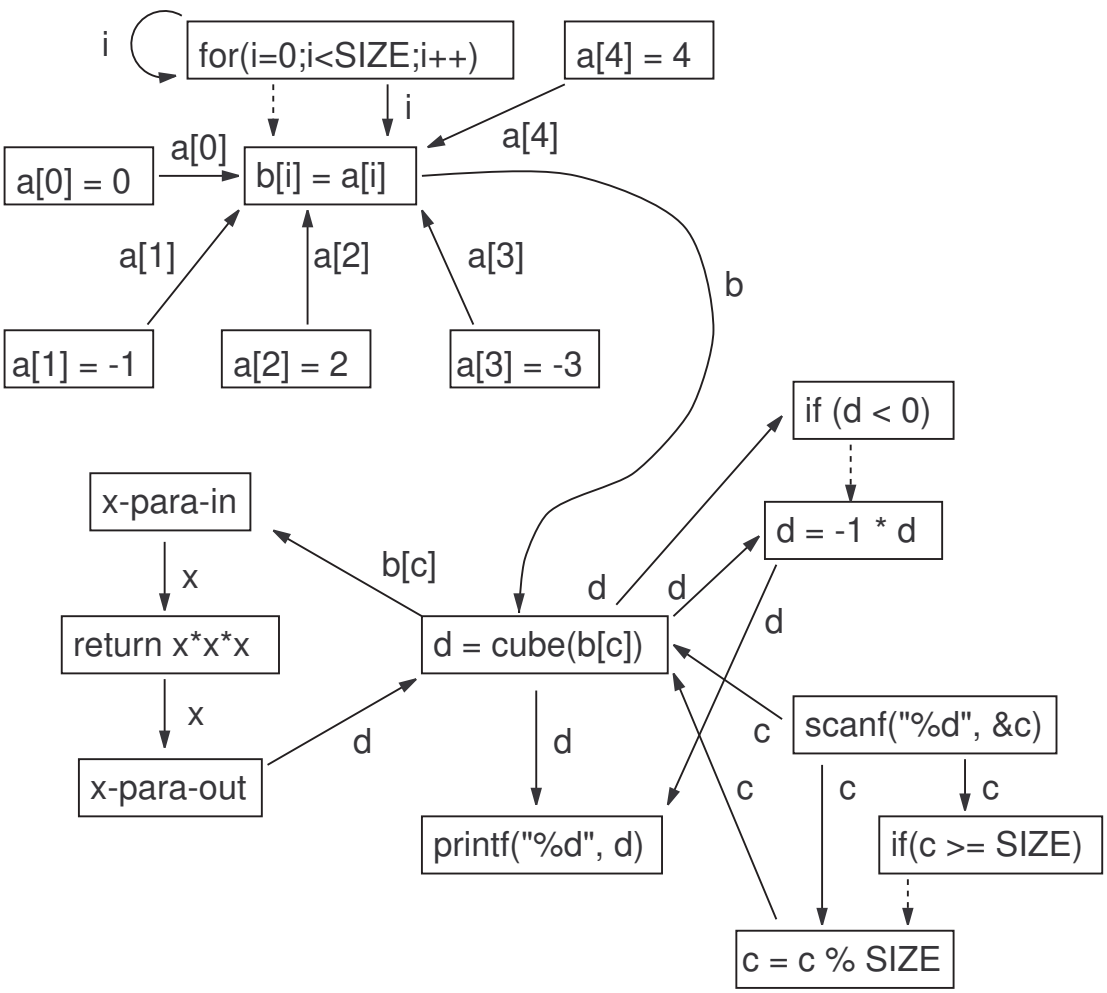

図 2: プログラム依存グラフ

```
 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:  }
```

```
 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:  }
```

図 3: 図 1 のプログラムの $(37, d)$ に関する静的スライス

2.2 動的スライス

2.2.1 動的スライスを計算するアルゴリズム

入力を与えてプログラムを実行した場合，実行された文の列を実行系列と呼ぶ．実行系列で p 番目に文の実行が行われた時点を実行時点 p と呼ぶ．動的スライスを計算するために必要となる動的依存関係情報を実行系列に与える．動的依存関係を以下の様に定義する．

1. データ依存関係（**Definition-Use**，**DU**）
   $p=\mathrm{Def}(q,w)$ とは，実行時点 q より前で最後に変数 w を定義した実行時点が p であることを表す．最後に定義されたとは，$p<r<q$ であるような全ての実行系列 r において，変数 w が定義されていないことをいう．また，実行時点 p において実行された文を $\mathrm{Ins}(p)$ で表し，実行時点 p に対して，$\mathrm{Def}(p)$ は実行時点 p における文 $\mathrm{Ins}(p)$ の実行により定義された変数，$\mathrm{Use}(p)$ は実行時点 p における文 $\mathrm{Ins}(p)$ の実行により使用された変数の集合を表す．実行時点 p から実行時点 $q(p<q)$ に対してデータ依存関係 $\mathrm{DU}(p,q)$ があるとは，ある変数 $w\in\mathrm{Use}(q)$ が存在して，$p=\mathrm{Def}(q,w)$ の場合をいう．データ依存関係 $\mathrm{DU}(p,q)$ は，実行時点 p で設定したある変数 w の値が実行時点 q で参照されたことを示している．変数 $w\in\mathrm{Use}(q)$ に関してデータ依存関係があることを明示する場合には，$\mathrm{DU}w(p,q)$ と表すこととする．

2. 制御依存関係（**Test-Control**，**TC**）
   まず，命令 t に対して，命令の集合 $\mathrm{CtlExec}(t)$ を次のように定義する．

   $$\mathrm{CtlExec}(t)=\{\text{ 命令 }s|\mathrm{CD}(s,t)\}$$

   ただし，命令 t がループ命令の場合には命令 t も $\mathrm{CtlExec}(t)$ に含める．
   実行時点 p から実行時点 $q(p<q)$ に対して制御依存関係 $\mathrm{TC}(p,q)$ があるとは，

   $$p=\max\{\text{ 実行時点 }i|i<q\text{ かつ }\mathrm{Ins}(i)\in\mathrm{CtlExec}(\mathrm{Ins}(q))\}$$

   の場合をいう．
   制御依存関係 $\mathrm{TC}(p,q)$ は，実行時点 q において命令 $\mathrm{Ins}(q)$ が実行されたことは，実行時点 p において実行された分岐命令あるいはループ命令 $\mathrm{Ins}(p)$ の制御移行に依存していたことを示している．

   また，実行系列 q における命令 $\mathrm{Ins}(q)$ が，関数，手続き内にある命令であるなら，その関数，手続きを呼び出した実行時点 q との間にも制御依存関係 $\mathrm{TC}(p,q)$ があるという．このとき実行時点 q における命令 $\mathrm{Ins}(q)$ は，関数，手続き呼び出し文である．

上記の依存関係を元に以下に示す依存関係情報を実行系列に与える．変数 v と実行時点 p との組 r を $\langle v,p\rangle$ と表す．ある実行時点 p の参照変数とその変数が定義された実行時点の組の集合を $VarREF(p)$

と書き．これを以下のように定義する．

$$VarREF(p) \equiv \{\ \langle v, q\rangle \mid \mathrm{DU}v(q,p)\}$$

また，ある実行時点 p と制御依存関係がある実行時点 q の集合を $\mathrm{Control}(p)$ と表す．

次にこれまでの手順により依存情関係報与えられた実行系列を元にプログラム P の動的スライスを計算する．まず，どの実行時点のどの変数に関して動的スライスを計算するかを決定する．その実行時点を動的スライスのスライス基準といい $C = (x, r, V)$ と表す．x, r, V は以下のとおりである．

- x はプログラム P に与えた入力．
- r はプログラム P に入力 x を与えたときの実行系列における実行時点．
- V はプログラム P 内の変数の部分集合．

プログラム P のスライス基準 $C = (x, r, V)$ に関する動的スライスは以下の手順により求められる．

1. DS $\leftarrow \phi$
   CalcDsObj $\leftarrow \phi$
   CalcDsObj を動的スライスを計算する対象となる実行時点の集合とする．

2. DS $\leftarrow \{\mathrm{Ins}(r)\}$
   CalcDsObj $\leftarrow$ { 実行時点 q | ある変数 $v \in$ V に対して q=Def(r, v)}

3. $p \in$ CalcDsObj
   CalcDsObj $\leftarrow$ { 実行時点 q | $VarREF(p) \cup \mathrm{Contorol}(p)\} \cup$ CalcDsObj
   DS $\leftarrow$ DS $\cup \{\mathrm{Ins}(p)\}$
   CalcDsObj $\leftarrow$ CalcDsObj - $\{p\}$

4. CalcDsObj が空集合でないなら 3 を行う．CalcDsObj が空集合なら DS がプログラム P のスライス基準 $C = (x, r, V)$ に関する動的スライスとなる．

2.2.2 動的スライスの適用例

図 1 の C 言語で記述されたソースプログラムに対応する PDG は 図 4 のようになる．図 1 のプログラムに入力 2 を与えて実行した場合の実行系列を 図 5 に示す．また，スライス基準 $(\{2\}, 28, \{d\})$ 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．

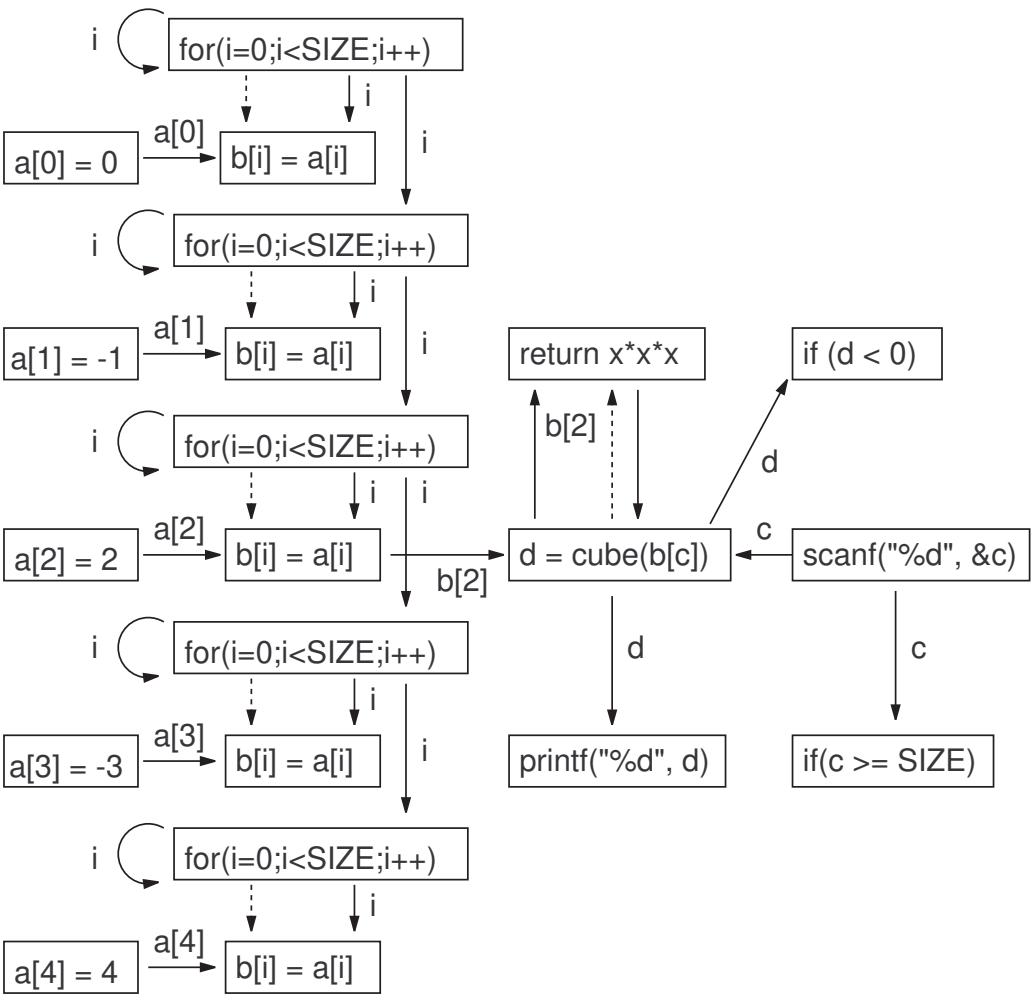

図 4: プログラム依存グラフ

```
 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:  }
```

```
 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:  }
```

図 6: 図 1 のプログラムの $(\{2\}, 28, \{d\})$ に関する動的スライス

3 Dependence-Cache スライス

静的スライス計算法は，スライスに含まれるべき文が排除されることを避けるために，全ての可能な入力データを考慮することになる．このため，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) もスライスに含んでしまうという欠点がある．対して，動的スライス計算法は，特定の入力データのみを考慮し実行系列を対象に解析を行うことで，静的スライスと比較して，スライスサイズの減少 (正確性の向上) が期待できる．しかし，実行系列はソースプログラムと比較して膨大な量になってしまう場合が多く，その保存と解析に多くの空間，時間コストを必要としてしまう．つまり，スライスの計算に必要なコストとスライスサイズ (正確性) はトレードオフの関係にある．

DC スライスは，動的にデータ依存関係を解析することで，配列の添字，ポインタの参照先を把握する．一方，制御依存関係は静的に解析し，実行系列を保存することはしないため動的スライス抽出に比べて実行時間の短縮が得られる．

3.1 計算手順

実行中に，ある文 s である変数 v が参照された時，v がどの文 (t) で定義されたかが分かれば，$DD(t,v,s)$ どいうデータ依存関係があることが分かる．逆に言えば，v を定義してる文さえ分かればデータ依存関係を知ることができる．

そこで，全ての変数に対して，その値を定義したのはどの文か，という情報を持たせておき，その変数の参照があった場合には，その情報からデータ依存関係を把握する．

また，得られたデータ依存関係は各文に持たせる．各文 s は集合 $DDS(s)$ を持つとする．この $DDS(s)$ の要素は，2 つの要素からなる組であり，(“依存関係の原因となる変数 v”，“s がデータ依存している文”) となっている．

ある実行時点において，変数 v を最後に定義した文を $DefS(v)$ とする．ここで v は，動的に生成される変数も含めた全ての変数を考える．配列では，各要素にも $DefS$ を考える．

次にアルゴリズムを示す．

(1) ある文 s を実行する前に，$DDS(s) = \phi$ とする．

(2) 入力データを与えてプログラムの実行を行う．今，文 s が実行されたとする．

- 文 s で変数 v が参照[1] された場合，$DDS(s) \leftarrow DDS(s) \cup (v, DefS(v))$ ．とする．
- 文 s で変数 v が定義された場合，$DefS(v) = s$ とする．

この作業によって得られた $DDS(s)$ は，全てのデータ依存関係を表しており，$DDS(s) = \{(v,t)|DD(t,v,s)$ が成り立つ $\}$ となっている．

[1] ポインタ変数が出現した場合，何が参照されたかを判断する際に注意が必要となる．例えば，代入文の右辺や出力文などに $**v$ という 2 階のポインタが現れた場合，v，$*v$，$**v$ が参照されたと考える．また，代入文の左辺に現れた変数は普通参照されたとは考えないが，例えば $*v$ という一階のポインタが現れた場合は，v を参照したと考える．

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

図 7: 配列を含むプログラム

表 1: 図 7 のプログラムにおける，各実行時点でのキャッシュの推移

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

(3) 集合 DDS を使って PDG を構築する．文 s の解析において，

- $DDS(s) \neq \phi$ の間，次の操作を繰り返す．
  - $DDS(s) \leftarrow DDS(s) - \{(v,t)\}$ ．
  - データ依存辺 $t \xrightarrow{v} s$ を引く．
- s が制御文であれば，その制御文内の全ての文 t に対して，制御依存辺 $s \dashrightarrow t$ を引く．

3.2 適用例

3.2.1 配列を含むプログラム

図 7 のプログラムに対して，動的データ依存関係解析を行う．入力に 0 を与えた場合を考える．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 に表す．

文 $1, 2, 3$ では，それぞれ変数 $a[0], a[1], a[2]$ が定義されているので，文 3 を実行した時点で $C(a[0]) = 1, C(a[1]) = 2, C([2]) = 3$ となる．文 4 では，c を定義しているので，$C(c) = 4$ となる．文 5 では，$a[0]$ を使用している．文 5 の実行直前には $C(a[0]) = 1$ であるので，文 1 から文 5 へ $a[0]$ に関するデータ依存辺を引く．同様に c を使用しており，文 4 から文 5 へ c に関するデータ依存辺を引く．また，b を定義しているので，$C(b) = 5$ となる．

3.2.2 ポインタを利用するプログラム

図 8 のプログラムに対して，動的データ依存関係解析を行う．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 2 に表す．

```
1  a=2;
2  b=1;
3  c=&a;
4  d=&c;
5  *c=5;
6  **d=b;
7  printf("%d",a);
8  printf("%d",**d);
```

図 8: ポインタを利用するプログラム

表 2: 図 8 のプログラムにおける，各実行時点でのキャッシュの推移

実行文	a	b	c	d
1	1	-	-	-
2	1	2	-	-
3	1	2	3	-
4	1	2	3	4
5	5	2	3	4
6	6	2	3	4
7	6	2	3	4
8	6	2	3	4

文 $1, 2, 3, 4$ では，それぞれ変数 a, b, c, d が定義されているので，文 4 を実行した時点で $C(a) = 1, C(b) = 2, C(c) = 3, C(d) = 4$ となる．

文 5 では，c を使用している．文 5 の実行直前には $C(c) = 3$ であるので，文 3 から文 5 へ c に関するデータ依存辺を引く．また，$*c$ を定義しているので，$C(*c) = 5(C(a) = 5)$ となる．

文 6 では，b ，d ，$*d$ を使用しているので，文 $2, 4, 3$ から文 6 へデータ依存辺を引く．さらに，文 6 では $**d$ を定義しているので，$C(**d) = 6(C(a) = 6)$ となる．

文 7 では，a を使用している．文 7 の実行直前には $C(a) = 6$ であるので，文 6 から文 7 へ a に関するデータ依存辺を引く．

文 8 では，$d, *d, **d$ を使用しているので，文 $4, 3, 6$ から文 8 へデータ依存辺を引く．

3.3 他のスライス手法との比較

静的スライス，動的スライス，DC スライスの違いを 表 3 に示す．

解析手法の違いにより，一般に以下の様な性質となる．これらの性質は [3] において，実験によって実証されている．

スライスサイズ: 静的 $\geq$ DC $\geq$ 動的

実行前解析時間: 静的 $>$ DC $>$ 動的

実行時間: 動的 $>$ DC $>$ 静的

表 3: スライス手法の違い

	静的スライス	動的スライス	DC スライス
制御依存関係解析	静的	動的	静的
データ依存関係解析	静的	動的	動的
PDG 節点	文	実行系列	文

また，静的スライス (図 3), 動的スライス (図 6) との比較のため， 図 1 のプログラムの，スライス基準 $(37, d)$ に関する DC スライスを 図 9 に，比較のため元のソースプログラムと併せて示す．入力には，動的スライスと同じく 2 を与えた．

```
 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:  }
```

```
 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:  }
```

図 9: 図 1 のプログラムの $(37, d)$ に関する DC スライス

4　オブジェクト指向言語を対象としたスライス

オブジェクト指向言語には，従来の手続き型言語にはないクラス，継承，ポリモーフィズムなどの新しい概念が導入されており，既存の手続き型言語を対象としたプログラムスライス計算手法をそのまま用いても正確なスライスが得られない．そこで，オブジェクト指向言語への対応を試みた静的スライス [9]，動的スライス [16] が提案されている．

4.1　静的スライスの適用と問題点

静的スライスは 2.1 で紹介した手法をもとに PDG の拡張を行う．ただし，前提として以下が挙げられている．

- 全てのエイリアスはエイリアス解析アルゴリズムにより解決できる
- オブジェクトのデータメンバはメソッドを通してのみアクセスされる
- 静的データメンバは大域変数として扱う

PDG の拡張は以下の項目について行う．

特殊節点の追加：　メソッド呼びだしを行うオブジェクトのデータメンバを特殊節点に追加する

特殊節点のデータメンバに関する拡張：　メソッドの引数がオブジェクトの際には，その特殊節点を木構造に拡張する．木構造の根はオブジェクト自身を表し，子はそのオブジェクトのデータメンバを表す．木構造の各辺は，オブジェクトとデータメンバ間に存在するデータ依存関係を表す．木構造の子がオブジェクトである場合は，さらにその子を木構造に拡張する．この拡張を再帰的に全ての葉が基本型となるまで繰り返すが，収束しない場合を考慮し，k-limiting を用いる (木の深さが k となるまで拡張を行う)．

特殊節点の継承されたオブジェクトに関する拡張：　オブジェクトの継承関係は木構造で表される．特殊節点が継承オブジェクトを表す際には，その特殊節点を木構造に拡張する．木構造の根は継承されたオブジェクトを表し，子はその取りうる型を表す．

この手法は，オブジェクト指向言語 (C++) を対象に正確なスライスを得ることを目的としている．ここで前提として，スライスにおける正確性の意味について以下の二つを挙げる．

- スライス基準に影響を与えうる文は全てスライスに含まれる
- 上記を前提に，スライス基準に真に影響を与えない文はスライスに含めない

本稿で用いる正確性とは後者の意味を指す．

この手法は，前者の正確性を満たすこと，つまり，スライス基準に影響を与えうる文を全てスライスに含めることを目的とし，後者の正確性，つまり，スライス基準に影響を与えうる文は全てスライスに含むが，真に影響を与えない無駄な文をスライスに含めないことについては考慮されていない．

静的な解析では，オブジェクトの取りうる型の限定，呼び出されるメソッドの限定を行うが，一意に決定できるわけではないために正確性は低くなる．

4.2 動的スライスの適用と問題点

動的スライスは 2.2 で紹介した手法をオブジェクト指向言語 (C++) に適用している．

動的スライスは実行時に決定される参照型の指すインスタンスを容易に調べられるため，正確な解析が行える．Java においてオブジェクトは全て参照により操作されるため，オブジェクトへの参照を正確に解析することは重要である．

[16] では，解析コストについては述べられていないが，手続き型言語を対象とした動的スライス抽出には，非常に大きなコストがかかり，現実的でないことがわかっている [3]．同様の実行系列を保存する手法を用いるこの手法にも，大きな解析コストがかかることが推測できる．

5　静的情報と動的情報を用いた Java スライス手法

現在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, C++等いわゆるオブジェクト指向言語の利用が高まっている．特に近年利用が高まっている Java を対象にしたスライス抽出を考える．

5.1　提案手法

4 において述べたが，正確性の観点から見た場合，動的スライスの方が静的スライスより優れている．しかし，動的スライスは手続き型言語を対象とした動的スライス同様に実行系列を保存するため，解析に多大なコストがかかると考えられる．

そこで，提案手法では，DC スライスの Java への適用を考える．DC スライスはデータ依存関係解析を動的に行うために，オブジェクトへの参照等の実行時に決定される要素を多く含む Java の解析に向いていると言える．また，実行系列を保存しないこと，及び制御依存関係解析を静的に行うことで解析コストを抑えることが期待できる．

次に DC スライスの Java への適用方針について述べる．

5.1.1　方針

Java は以下のような実行時に決定される要素を持つ．

配列の添字：　配列の添字は実行時に決定される場合があり，その場合アクセスされる配列の要素は，実行時に決定される

オブジェクトへの参照：　参照型変数が参照するオブジェクトは実行時に決定される

動的束縛：　実行中のオブジェクトのクラスに基づいて適切なオーバーライドメソッドが選択される

例外：　全ての文の実行で例外の発生する可能性があり，制御はその種類の例外を処理できる最初の catch 節に移る

マルチスレッド：　マルチスレッドプログラムの制御の流れは一意に決定されない

これらの要素を静的に解析する場合，充分な情報が得られないため正確性に限界がある．

DC スライスは，データ依存関係解析を動的に行うことで，配列の要素を区別した解析，ポインタの解析を行うことができる．本手法でも，データ依存関係解析を動的に行うことで，DC スライスと同様に配列の要素を区別した解析を実現する．また，ポインタに相当する参照型変数の解析も正確に行うことができる．

また，オブジェクト指向の基本概念として以下が挙げられる．

- オブジェクトとは，データとそれを操作するコード (メソッド) の組み合わせである．

- カプセル化によりデータへのアクセスはメソッドを通して行う．
- 動的束縛では，実行中のオブジェクトのクラスに基づいて適切なオーバーライドメソッドを選択する．

つまり，オブジェクト間のデータの操作にはメソッド呼び出しが生じ，呼び出されるメソッドは実行時に決定される．これより，実行時に決定されるメソッド呼び出しが頻繁に起こりうることがわかる．そこで，従来の DC スライスは，全ての制御依存関係解析を静的に行っているが，本手法では，データ依存関係解析に加えメソッド呼び出しに関する制御依存関係解析も動的に行うことで呼び出されるメソッドを一意に決定し，解析の正確性を高める．

例外，マルチスレッドについては 7 で述べる．

5.1.2 アルゴリズム

図 10 に計算手順を示す．

(1), (2) で静的な制御依存関係解析を行い，PDG を構築する．(3) ではデータ依存関係解析とメソッド呼び出しに関する制御依存関係の動的な解析を行い，PDG を完成させる．動的データ依存関係解析の計算手法は 5.1.3 で示す．(4), (5), (6) では，スライス基準に対応する節点から PDG をたどり，スライスを抽出する．

5.1.3 動的データ依存関係解析アルゴリズム

制御依存関係，データ依存関係の動的な解析の計算手順を 図 11 に示す．

各変数に対応するキャッシュはその変数が生成されるときに逐次生成する．同じクラスから生成された複数のインスタンスはそれぞれ独立にインスタンス変数を保持する．同様に，各インスタンス変数のキャッシュは各インスタンス変数が生成される際に生成され，それらは独立して保持される．つまり，インスタンスごとに独立した解析を行う．

5.2 適用例

図 12 のプログラムを入力 -1 について実行した場合の各実行時点におけるキャッシュC の推移を表 4 に表す．ただし，クラス $Base$，クラス $Derived$ 内のローカル変数 i については省略した．

また，スライス基準 $(16, c)$ に関するスライスを 図 13 に示す．

表 4: 図 12 のプログラムにおける，各実行時点でのキャッシュの推移

実行文	$b1$	$b2$	c	i	$b1.a$	$b2.a$	$args$	$args[0]$
4	-	-	-	-	-	-	-	-
5	5	-	-	-	-	-	-	-
6	5	-	-	-	-	-	6	6
7	5	-	7	-	-	-	6	6
8	5	8	7	-	-	-	6	6
31	5	8	7	-	-	-	6	6
21	5	8	7	-	-	-	6	6
22	5	8	7	-	-	22	6	6
9	5	8	7	9	-	22	6	6
10	5	8	7	9	-	22	6	6
11	11	8	7	9	-	22	6	6
21	11	8	7	9	-	22	6	6
22	11	8	7	9	22	22	6	6
14	11	8	14	9	22	22	6	6
23	11	8	14	9	22	22	6	6
24	11	8	14	9	22	22	6	6
15	11	8	14	9	22	22	6	6
26	11	8	14	9	22	22	6	6
27	11	8	14	9	22	27	6	6
16	11	8	14	9	22	27	6	6
17	11	8	14	9	22	27	6	6

入力

P: スライス対象 Java プログラム

I: P への入力

(s_c, v_c): スライス基準

一時記憶

PDG_D: P の I についての実行時の PDG

N: 節点の集合

C: 節点

出力

OUT: P の I についての実行の出力

S: P の I についての実行時, (s_c, v_c) に関するスライス

アルゴリズム

(1) P の全ての文または条件節 s について，PDG_D の節点 $V(s)$ を作成する

(2) P の全ての条件文 s について，その条件節を cnd，その中身の文を stm とすると，制御依存辺 $V(cnd) \dashrightarrow V(stm)$ を PDG_D に加える

(3) I についての P の実行が終了するまで以下を繰り返す (I について実行する次の文を s とする)

(a) s について，動的データ依存関係解析を行う

(b) s がメソッド宣言文ならば

(i) $V = C$

(ii) 制御依存辺 $V \dashrightarrow V(s)$ を PDG_D に加える

(c) s がメソッドを呼び出すならば
$C = V(s)$

(d) I について，s を実行する

(4) $S = \{V(s_c)\}, N = \phi$

(5) $N =$ { 節点 $n | n \xrightarrow{v} s_c$ } $\cup$ { 節点 $m | m \dashrightarrow s_c$ }

(6) $N \neq \phi$ の間，以下を繰り返す

(a) 節点 $n \in N$ を 1 つ選ぶ

(b) $S = S \cup n$

(c) $N = N \cup \{m | m \notin S \wedge (m \xrightarrow{w} n \vee m \dashrightarrow n)\}$
ただし，w は任意の変数名

図 10: Java スライシングアルゴリズム

入力

s: スライス対象 Java プログラム P 中の文

I: P への入力

PDG_D: P の I についての実行時の PDG

一時記憶

$C(v)$: 変数 v のキャッシュ

出力

OUT: P の I についての実行の出力

PDG_D: P の I についての実行時の PDG

アルゴリズム

文 s において，変数 v が

(1) 宣言された場合

(a) $C(v)$ を作成

(b) $C(v) = V(s)$

(2) 定義された場合

$C(v) = V(s)$

(3) 参照された場合

v に関するデータ依存辺 $C(v) \xrightarrow{v} V(s)$ を PDG_D に加える

図 11: 動的データ依存関係解析アルゴリズム

```
 1: import java.util.*;
 2: import java.io.*;
 3:
 4: class Main {
 5:     static Base b1;
 6:     public static void main(String[] args) throws IOException {
 7:         int c;
 8:         Base b2 = new Derived();
 9:         int i = Integer.parseInt(args[0]);
10:         if (i < 0)
11:             b1 = new Base();
12:         else
13:             b1 = new Derived();
14:         c = b1.m(i);
15:         b2.set(c);
16:         System.out.println(c);
17:         System.out.println(b1.a);
18:     }
19: }
20:
21: class Base {
22:     public int a = 10;
23:     public int m(int i) {
24:         return (a - i);
25:     }
26:     public void set(int i) {
27:         a = i;
28:     }
29: }
30:
31: class Derived extends Base {
32:     public int m(int i) {
33:         return (a + i);
34:     }
35: }
```

図 12: プログラム

```
 1:  import java.util.*;
 2:  import java.io.*;
 3:
 4:  class Main {
 5:      static Base b1;
 6:      public static void main(String[] args) throws IOException {
 7:          int c;

 9:          int i = Integer.parseInt(args[0]);
10:          if (i < 0)
11:              b1 = new Base();


14:          c = b1.m(i);

16:          System.out.println(c);

18:      }
19:  }
20:
21:  class Base {
22:      public int a = 10;
23:      public int m(int i) {
24:          return (a - i);
25:      }



29:  }
```

図 13: 図 12 のプログラムのスライス基準 $(16, c)$ に関するスライス

6 本手法の実装

提案手法の実装方針としてインタプリタ方式とプリプロセッサ方式が考えられる．

インタプリタ方式は，インタプリタが実行，解析を行う．実行環境自体が解析を行うため，あらゆる実行時情報を取得可能である．よって，スライス抽出のための解析以外の様々な動的解析も容易に行うことができるため，デバッグに有用な情報を容易に取得できる．問題点としては，Java Virtual Machine の実装と同規模の非常に大きな手間がかかることが挙げられる．また，インタプリタであるので実行速度が遅い．

プリプロセッサ方式は，Java のソースプログラムをコンパイルする前に，そのプログラム自身の動的解析を行う命令を付加するプリプロセッサによる実装である．解析命令を付加したプログラムをコンパイル，実行することで，通常の実行を行いながら解析も自動的に行われる．実行に関しては，通常の実行と同様に行えるため，Just-in-Time コンパイラ等による高速実行が可能である．

今回は，実現容易性，実行速度で有利なプリプロセッサ方式の実装を選択した．

6.1 実装概要

スライス対象である Java プログラムは，様々なプラットフォーム上で開発される可能性がある．提案手法の実現であるスライスツールも様々なプラットフォーム上で動作するよう，実装言語として Java を用いた．

スライスエンジン部

入力: Java ソースファイル (Java1.1.8 以下)

出力: 自身の解析と通常の実行を行うバイトコード

概要: 入力ファイルを構文解析し，プログラム中の各文で定義，参照される変数を見つける．それらの解析を行う命令を入力ファイルに付加する．入力プログラムで使用されるクラスを格納するソースファイルが，JDK[17] の javac が行うクラス検索と同じ検索法で見つかった場合，それらのソースファイルについても解析，命令の付加を行う．全ての見つかったソースファイルについて命令の付加を行った後に，javac を用いてそれらのバイトコードを生成する．

クラス数: 33

行数: 約 8,000

ユーザインタフェース部

概要: スライスを抽出するための操作を総合的に管理する

- Java ソースファイルを読み込み表示を行う

- 読み込んだソースファイルに対し，外部プログラム (スライスエンジン部，JDK の java) を呼び出すことで，解析命令の付加，生成されたバイトコードの実行を行う．
- スライスの表示
  - 解析命令を含むプログラムの実行出力であるプログラム依存グラフを受け取る
  - ユーザからのスライス基準の指定を受け取る
  - グラフ探索を行い，スライスを抽出し，表示に反映させる

クラス数: 28

行数: 約 2,000

実装概要を 図 14 に示す．

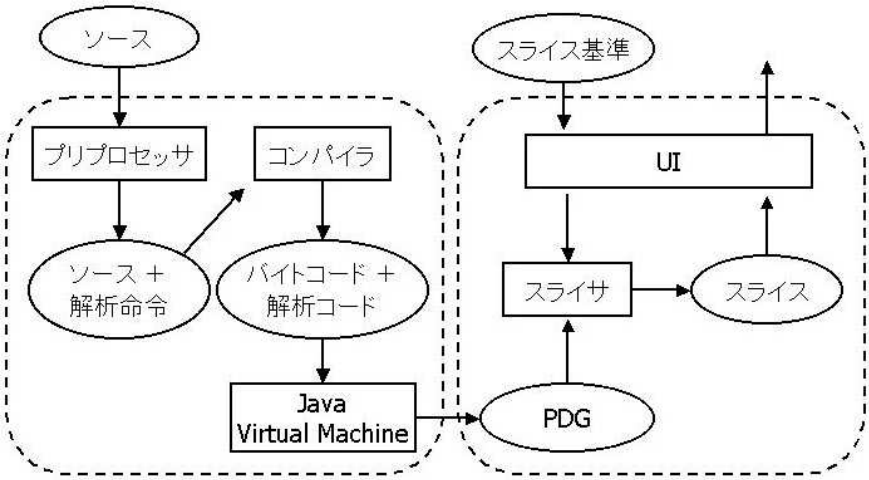

図 14: 実装概要

6.2 キャッシュの実装

各変数のキャッシュを以下のように実装する．

- 各クラスに現れるインスタンス変数 v に対してキャッシュ$C(v)$ をそのクラスのインスタンス変数として用意する．
- 各クラスに現れる静的変数 v に対してキャッシュ$C(v)$ をそのクラスの静的変数として用意する．

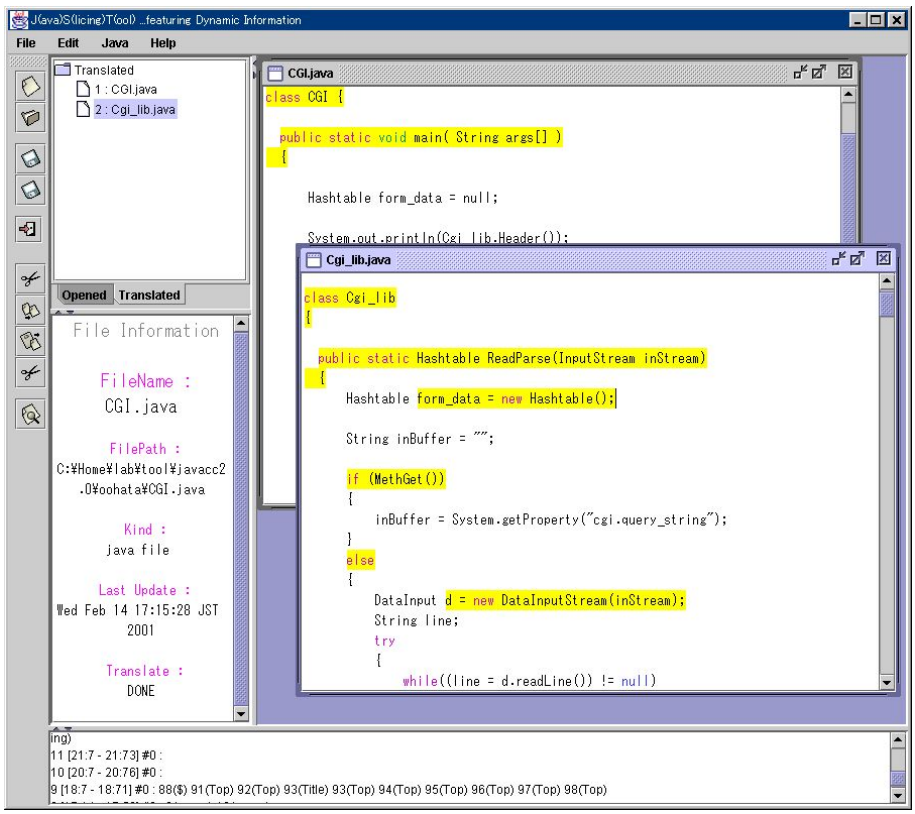

図 15: Java Slicing Tool

- 各クラスに現れるローカル変数 v に対してキャッシュ$C(v)$ を v が存在するスコープ内のローカル変数として用意する．

これにより，あるクラスの複数のインスタンスが生成された際に，それぞれのインスタンスが各インスタンス変数 v を独立して保持するのと同様に，それぞれのインスタンスが各キャッシュ$C(v)$ を独立に保持する．つまり，それぞれのインスタンスについて独立した解析を行うことができる．また，変数 v が継承される場合，同様にキャッシュ$C(v)$ も継承される．

6.3 評価

前節で述べた Java スライスツールを用いて本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. スライスサイズの比較
2. 実行時間
3. 実行時使用メモリ

1. では，静的，動的スライス抽出手法との比較を行った．実装は，本手法のみ行ったため，本手法のスライスはツールを用いて求め，他の手法のスライスは手計算で求めた．

2, 3 では，解析命令を付加する前の実行と付加したあとの実行を比較することで，スライスを得るために必要な解析コストを計測した．

スライス対象プログラムの概要を 表 5 に示す．

P1 は入力としてファイルを読み込み，その内容を解析し，しかるべき HTML ファイルを出力する CGI プログラムである．

P2 は GUI を持つ，マウス操作によるペイントアプリケーションである．このプログラムは，JDK1.3.0_01 に付属のデモプログラム DrawTest の 3 つのクラスを実験用に変更したものである．

表 5: 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	2	0	223
P2	3	7	226

6.3.1 スライスサイズ

オブジェクト指向言語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本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．各種法ともに，スライス基準に影響を与えうる文は全てスライスに含まれるため，得られたスライスサイズが小さいほど正確である (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) と言える．

計測値は，任意に定めた 3 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，本手法のスライスはツールを用いて求め，他の手法のスライスは手計算で求めた．計測値を 表 6 に示す．

表 6: 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	本手法
P1-スライス基準 1	26 (11.7%)	15 (6.7%)	15 (6.7%)
P1-スライス基準 2	83 (37.2%)	27 (12.1%)	27 (12.1%)
P1-スライス基準 3	37 (16.6%)	24 (10.8%)	24 (10.8%)
P2-スライス基準 1	48 (21.2%)	14 (6.2%)	14 (6.2%)
P2-スライス基準 2	45 (19.9%)	12 (5.3%)	12 (5.3%)
P2-スライス基準 3	25 (11.1%)	17 (7.5%)	17 (7.5%)

6.3.2 解析コスト

解析命令を付加する前の実行 (通常の実行) と付加したあとの実行を比較することで，スライスを得るために必要な解析コストを計測した．実行環境を 表 7 に示す．

表 7: 実験環境

CPU	Celeron-500MHz
メモリ	128MB
OS	Windows98 Second Edition
Java	JDK 1.3.0_01 HotSpot

計測値は，実行を 10 回行った際の平均値である．実行時間の計測値を 表 8 に示す．実行時の最大使用メモリ量の計測値を 表 9 に示す．ただし，P2 については対話型プログラムであり実行時間の正確な計測が困難なため，実行時間を計測していない．

表 8: 実行時間 [ms]

プログラム	通常実行	解析付き実行	解析付き実行/通常実行
P1	138	582	4.22

表 9: 使用メモリ量 [KByte]

プログラム	通常実行	解析付き実行	解析付き実行/通常実行
P1	478	645	1.35
P2	836	920	1.10

6.4 考察

表 6 より，本手法で得られるスライスは，静的スライスの約 29%から 67%のサイズであり，本手法では静的スライスより正確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，本手法によるスライスと静的スライスの差は少なかった．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，静的スライスの正確性がより低くなることが推測できる．本手法では，継承やオーバーライド，オーバーロードの数に関わらず正確性の低下は起きない．また，今回の実験においては，本

手法では動的スライスと等しいスライスが得られた．例えば，スライス対象プログラム中に 図 1 の文 14 から 22 と 31 のような命令がある場合，本手法は動的スライスより正確性が低くなる．

解析コストについて，本手法に対してのみ実装を行ったため，静的スライス，動的スライスに関するコストの計測は行えなかった．そこで，理論値により本手法と静的スライス，動的スライスの比較を行う．

ある入力 I についてプログラムを実行した時，実行された文の数を S_{exe}，実行されなかった文の数を S_{nonexe} とするとプログラム全体の文の数は $S_{exe} + S_{nonexe}$ で表される．プログラム中のクラスの数を c，メソッドの数を m，引数の数の最大値を P_{max} とする．

実行されたループの延べ数を l，各ループで実行された文の数を $Sloop_i(1 \leq i \leq l)$，ループの回数を $Nloop_i(1 \leq i \leq l)$ とする．実行された文の中でループに属さないものの数を SS_{exe} とすると，

$$S_{exe} = \sum_{i=1}^{l} Sloop_i + SS_{exe}$$

となる．実行系列の数を E とすると

$$E = \sum_{i=1}^{l} Sloop_i \times Nloop_i + SS_{exe}$$

となる．

プログラム全体で定義された変数の数を V，実行されなかった文における変数の使用回数を V_{nonexe} とする．また，実行されたループにおける変数の使用回数を $V_{loop_i}(1 \leq i \leq l)$，ループ以外での変数の使用回数を $V_{nonloop}$ とする．

本手法での PDG 節点の数を VTX とすると

$$VTX = S_{exe} + S_{nonexe}$$

である．

静的スライスでの PDG 節点の数を VTX_{static}，メソッド引数のための特殊節点の数を $Para$ とすると，

$$VTX_{static} = S_{exe} + S_{nonexe} + Para$$

となる．メソッド呼びだし文の数を M とすると $Para \subseteq O(M \times V \times P_{max} \times c)$ である．

動的スライスでの PDG 節点の数を $VTX_{dynamic}$ とすると，

$$VTX_{dynamic} = E$$

となる．

静的スライスの構築する PDG 辺の数は本手法より $Para$ 多く，そのための時間，空間コストがかかる．動的スライスの構築する PDG の辺の数は本手法による PDG 辺の数と等しい．

本手法ではキャッシュを用いるためキャッシュの作成，更新，参照のためのコストがかかる．キャッシュは V 作成され，$\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop}$ 回操作される．

1 個の PDG 節点の作成に必要な手間を C_{vtx}，メモリ量を M_{vtx}，1 個の PDG 辺の作成に必要な手間を C_{edg}，メモリ量を M_{edg}，1 個のキャッシュの操作に必要な手間を C_{ch}，1 個のキャッシュの作成に必要なメモリ量を M_{ch} とする．

本手法の解析にかかる手間 Ope は，

$$Ope = C_{vtx} \times VTX + C_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) + C_{ch} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right)$$

である．静的スライスの解析にかかる手間は，

$$\begin{aligned} & C_{vtx} \times VTX_{static} + C_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} + V_{nonloop} + V_{nonexe} \right) \\ = \quad & Ope + C_{vtx} \times Para + C_{edg} \times \left\{ \sum_{i=1}^{l} V_{loop_i} \times (1 - Nloop_i) + V_{nonexe} \right\} \\ & - C_{ch} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) \end{aligned}$$

である．動的スライスの解析にかかる手間は，

$$\begin{aligned} & C_{vtx} \times VTX_{dynamic} + C_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) \\ = \quad & Ope + C_{vtx} \times \left\{ \sum_{i=1}^{l} Sloop_i \times (Nloop_i - 1) - S_{nonexe} \right\} - C_{ch} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) \end{aligned}$$

である．

本手法の解析にかかるメモリ量 Mem は，

$$Mem = M_{vtx} \times VTX + M_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) + M_{ch} \times V$$

である．静的スライスの解析にかかるメモリ量は，

$$\begin{aligned} & M_{vtx} \times VTX_{static} + M_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} + V_{nonloop} + V_{nonexe} \right) \\ = \quad & Mem + M_{vtx} \times Para + M_{ede} \times \left\{ \sum_{i=1}^{l} V_{loop_i} \times (1 - Nloop_i) + V_{nonexe} \right\} - M_{ch} \times V \end{aligned}$$

である．動的スライスの解析にかかるメモリ量は，

$$\begin{aligned} & M_{vtx} \times VTX_{dynamic} + M_{edg} \times \left(\sum_{i=1}^{l} V_{loop_i} \times Nloop_i + V_{nonloop} \right) \\ = \quad & Mem + M_{vtx} \times \left\{ \sum_{i=1}^{l} Sloop_i \times (Nloop_i - 1) - S_{nonexe} \right\} - M_{ch} \times V \end{aligned}$$

である．

7　まとめと今後の課題

本研究では，オブジェクト指向言語である Java のプログラムに対して，動的・静的解析を用いてスライスを抽出する手法を提案した．本手法は，データ依存関係解析，メソッド間の制御依存関係解析を動的に行うことで，静的スライスに比べ正確性の高いスライスを抽出する．一方，メソッド間以外の制御依存関係解析は静的に行い，PDG 節点を各文に対して作成することで動的スライスに比べ解析コストを抑えた．また，本手法を Java スライシングツールとして実装し，その有効性を確認した．

今後の課題として，以下が挙げられる．

- マルチスレッドへの対応
- 例外への対応

Java ではマルチスレッドや例外も動的に決定される要素を含むため，これらに関する制御依存関係解析も動的に行うことで，正確な解析が行えると考えている．

謝辞

本論文の作成において，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 ソフトウェア科学分野井上克郎 教授に深く感謝致します．

本論文の作成において，常に適切な御指導を賜りました同 楠本真二 助教授に深く感謝致します．

本論文の作成において，常に適切な御指導を賜りました同 松下誠 助手に深く感謝致します．

本論文の作成において，適切な御助言を頂きました同 大畑文明氏に深く感謝致します．

本論文の作成において，提案手法の実装に御力添え頂きました同 藤井将人氏に深く感謝致します．

最後に，その他の面で様々な御指導，御助言を頂いた大阪大学大学院 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野井上研究室の皆様に深く感謝致します．

参考文献

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

[2] Agrawal H., DeMillo A. R. and Spafford H. E. : "Dynamic Slicing in the Presence of Unconstrained Pointers", *In Proceedings of the Symposium on Testing, Analysis and Verification*, pp. 60–73, October 1991.

[3] Ashida Y., Ohata F. and Inoue K. : "Slicing Methods Using Static and Dynamic Information", *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC'99)*, pp. 344–350, December 1999.

[4] Gosling J., Joy B. and Steele G., 村上 雅章 [訳] : "The Java 言語仕様" *Addison-Wesley Publishers Japan, Ltd.*, 1997.

[5] Harrold M. J., Rothermel G. and Sinha S. : "Computation of Interprocedural Control Dependencies", *Technical Report OSU-CISRC-7/97-TR36*, September 1998.

[6] Hirose K., Ohata F. and Inoue K. : "動的データ依存関係解析を用いたJavaプログラムスライス手法", 電子情報通信学会技術研究報告，*Vol. 100, No. 570*, pp. 65–72, 2001.

[7] Horwitz S. and Reps T. : "The Use of Program Dependence Graphs in Software Engineering", *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, 1992.

[8] Larsen L. D. and Harrold M. J. : "Slicing Object-Oriented Software", *Proceedings of the 18th International Conference on Software Engineering*, pp. 495–505, Berlin, March 1996.

[9] Liang D. and Harrold M. J. : "Slicing Objects Using System Dependence Graphs", *Proceedings of the IEEE International Conference on Software Maintenance*, Washington, D.C., November 1998.

[10] 佐藤, 飯田, 井上: "プログラムの依存解析に基づくデバッグ支援ツールの試作", 情報処理学会論文誌, *Vol. 37, No. 4*, pp. 536–545, 1996.

[11] Sinha S. and Harrold M. J. : "Analysis of Programs With Exception-Handling Construsts", *Proceedings of the IEEE International Conference on Software Maintenance*, Washington, D.C., November 1998.

[12] Steindl C. : "Static Analysis of Object-Oriented Programs", *9th ECOOP Workshop for PhD Students in Object-Oriented Programming*, Lisbon, Portugal, June , 1999.

[13] Steindl C. : "Program Slicing for Object-Oriented Programming Languages", *Dissertation for the degree of Doctor of Technical Sciences in Computer Science at the Johannes Kepler University Linz*, pp. 31–46, Austria, April 1999.

[14] 植田, 練, 井上, 鳥居: "再帰を含むプログラムのスライス計算法", 電子情報通信学会論文誌, *Vol. J78-D-I, No. 1*, pp. 11–22, 1995.

[15] Weiser M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.

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

[17] "Java2 Platform, Standard Edition", http://java.sun.com/j2se/

[18] "JavaCC", http://www.metamata.com/javacc/

[19] "Write CGI programs in Java - JavaWorld January 1997", http://www.javaworld.com/javaworld/jw-01-1997/jw-01-cgiscripts.html