修士学位論文

題目

静的情報と動的情報を用いた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
芦田佳行

平成 12 年 2 月 15 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 11 年度 修士学位論文

静的情報と動的情報を用いたプログラムスライス計算法

芦田佳行

内容梗概

プログラムのデバッグを支援するための方法として，プログラムスライス (以降，単にスライスと略す) が提案されている．スライスは静的スライス [19] と動的スライス [1] の 2 種類に大別される．スライスの計算には依存関係情報が必要となるが，静的スライスはソースプログラムのみを用いてそのプログラムの依存関係を解析する．必然的に全ての実行可能パスを考慮することになるため，抽出される依存関係の数が多くなり，その結果スライスサイズが大きくなってしまう．一方，動的スライスはプログラムを実行して得られる実行系列のみを利用して依存関係を解析するため，あるテストデータに特化した依存関係を抽出することになり，スライスサイズが抑えられる．ある特定のテストデータと密接に関連しているため，よりデバッグに有効であるということができる．しかし全ての実行系列を保存する作業は多くの時間，空間コストを必要とする．

本研究では，静的解析と動的解析を組み合わせた 3 つのスライス計算法を提案する．これらの手法は静的情報と動的情報を併用することによって，わずかな追加コストで動的スライスに近いスライスを得ることを目的としている．

(1) Statement-Mark スライス: 文の実行履歴を用いて，実行されなかった文 (スライスに不要と考えられる文) を取り除く．

(2) 部分解析法: 関数の実行履歴を用いて実行された文だけを静的に解析することで，依存解析のコストを削減する．

(3) D3 スライス: データ依存解析を動的に行うことで，より正確なデータ依存関係を抽出する．

また，これらの手法を既に試作しているスライシングツール [15] に実装し，従来手法と比較して有効性を確認する．

主な用語

プログラムスライス (Program Slice)
依存関係解析 (Dependence Analysis)
静的情報 (Static Information)
動的情報 (Dynamic Information)

目 次

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は M. Weiser [19] 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を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．

我々も実際に実験を行い，デバッグ，保守にプログラムスライスが有効であることを確認した [3, 12, 13]．

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する．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 H. Agrawal [1] は，よりデバッグに有効な手法として動的スライスを提案している．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．

本研究では，コストと正確性を考慮に入れた 3 つのスライス抽出手法を提案する．Statement-Mark スライスは，わずかなコストで得られる文の実行履歴を利用して，静的スライスのスライスサイズを減少させる手法である．部分解析法は，わずかなコストで得られる関数の実行履歴を利用して，スライスサイズだけでなく，依存関係解析のコストを削減する手法である．動的なデータ依存関係解

析法は，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．また，本手法をこれまでに試作したツールに実装し，従来手法との比較を行い，これらの手法の有効性を確認する．

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 [19] とは，プログラム中のある文 s におけるある変数 v に対して v の値に影響を与える全ての文，もしくは，s における変数 w の定義に対して w が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 (Program Slice) または単にスライス (Slice) と呼ぶ.

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 (Static Slice) 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 (Dynamic Slice) 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 [15] のアルゴリズムを用いる．このアルゴリズムは [10, 18] 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 略して 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．PDG は有向グラフであるので，そのたどり方にも二種類あり，有向辺を順方向にたどっていくことにより計算した静的スライスを Forward Slice，逆方向にたどっていくことにより計算した静的スライスを Backward Slice と呼ぶ.

- **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 (Data Dependence, 略して DD) 関係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 (Control Dependence，略して CD) 関係を表す.

DD 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 DD 関係があるという.

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 CD が存在するものとする．すなわち CD 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DD 関係が生じる． これらの DD 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 (中継節点と呼ぶ) 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，[18] 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 DD 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 DD 関係辺を示す．)

2.1.2 静的スライスの計算

上記の手順に従って 図 1 のプログラムにたいして静的スライスを実際に計算する．その際，作成された PDG を 図 2，その静的スライスを 図 3， 図 4 に示し，比較のため 図 3， 図 4 には元のプログラムを一緒につけておく．

図 3 は 図 1 の 36 行目の文 `writeln(g)` に対応する PDG の節点の変数 `g` に関する Backward Slice である．このスライスには変数 `l` を求めるための関数 `lcm` や関数 `lcm` を呼び出す文，変数 `l` を出力する文などは静的スライスに含まれていない．

図 4 は， 図 1 の 24 行目の文の `w:=m mod n` に対応する PDG の節点の変数 `w` に関する Forward Slice である．このスライスには手続き `swap` や 32 行目の入力文 `readln(x,y)` などはスライスに含まれていない．

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;
 4  procedure swap(var a,b:integer);
 5    var     temp:integer;
 6    begin
 7      temp:=a;
 8      a:=b;
 9      b:=temp;
10    end;
11  function lcm(a,b:integer):integer;
12    var     c:integer;
13    begin
14      c:=gcd(a,b);
15      lcm:=(a div c)*(b div c)*c
16    end;
17  function gcd;
18    var     w:integer;
19    begin
20      if m < n then begin
21        swap(m,n);
22      end;
23      while n <> 0 do begin
24        w:=m mod n;
25        m:=n;
26        n:=w;
27      end;
28      gcd:=m;
29    end;
30  begin
31    writeln('Input x and y');
32    readln(x,y);
33    writeln('x=',x,' y=',y);
34    g:=gcd(x,y);
35    l:=lcm(x,y);
36    writeln('gcd=',g);
37    writeln('lcm=',l);
38  end.
```

図 1: プログラム

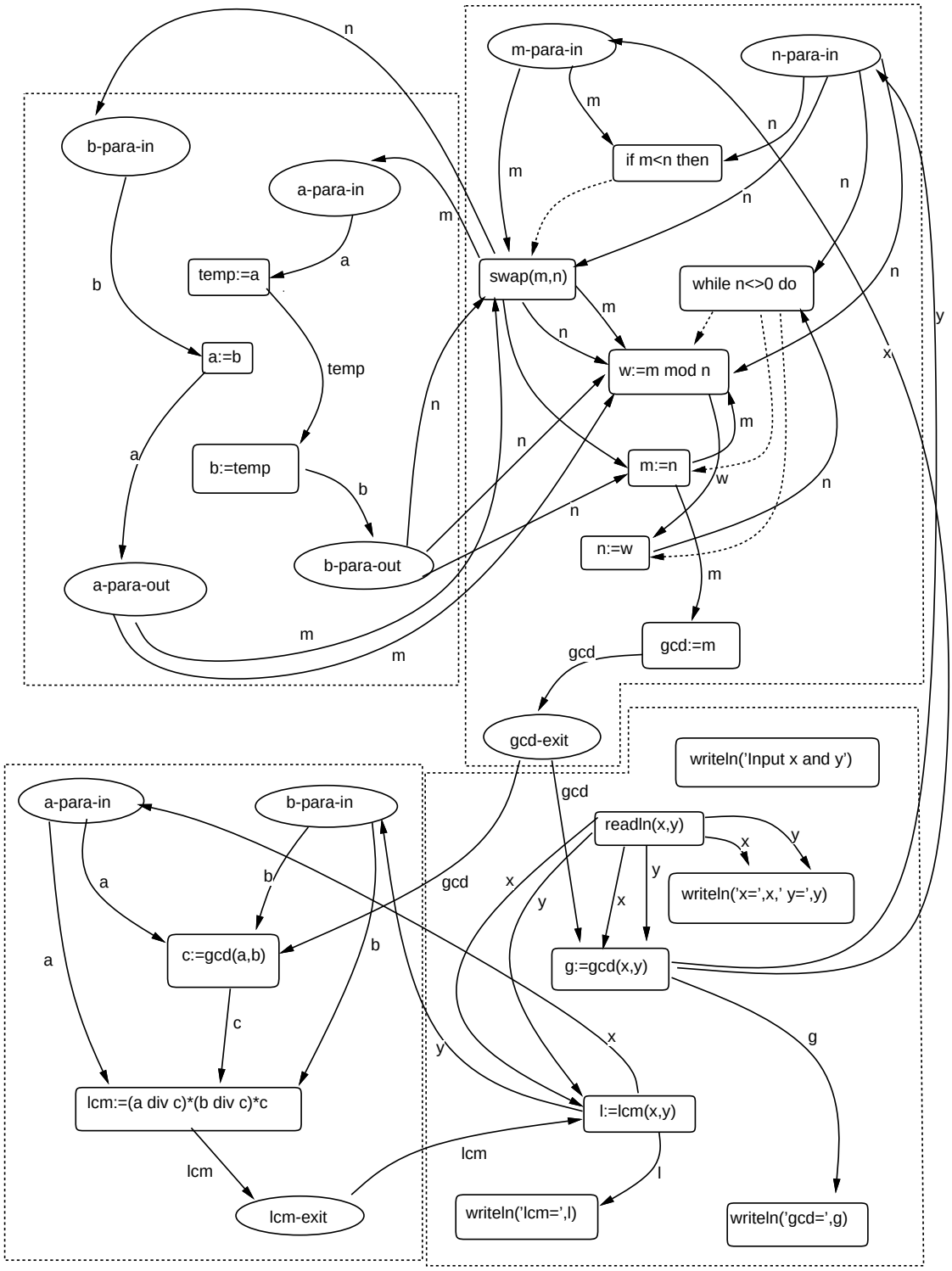
m-para-in
n-para-in
b-para-in
a-para-in
if m<n then
temp:=a
swap(m,n)
while n<>0 do
a:=b
w:=m mod n
b:=temp
m:=n
n:=w
b-para-out
a-para-out
gcd:=m
gcd-exit
writeln('Input x and y')
readln(x,y)
writeln('x=',x,' y=',y)
g:=gcd(x,y)
writeln('gcd=',g)
a-para-in
b-para-in
c:=gcd(a,b)
lcm:=(a div c)*(b div c)*c
lcm-exit
l:=lcm(x,y)
writeln('lcm=',l)
n
m
a
b
temp
w
c
x
y
g
l
gcd
lcm

図 2: PDG の例

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;
 4  procedure swap(var a,b:integer);
 5    var     temp:integer;
 6    begin
 7      temp:=a;
 8      a:=b;
 9      b:=temp;
10    end;
11  function lcm(a,b:integer):integer;
12    var     c:integer;
13    begin
14      c:=gcd(a,b);
15      lcm:=(a div c)*(b div c)*c
16    end;
17  function gcd;
18    var     w:integer;
19    begin
20      if m < n then begin
21        swap(m,n);
22      end;
23      while n <> 0 do begin
24        w:=m mod n;
25        m:=n;
26        n:=w;
27      end;
28      gcd:=m;
29    end;
30  begin
31    writeln('Input x and y');
32    readln(x,y);
33    writeln('x=',x,' y=',y);
34    g:=gcd(x,y);
35    l:=lcm(x,y);
36    writeln('gcd=',g);
37    writeln('lcm=',l);
38  end.
```

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;
 4  procedure swap(var a,b:integer);
 5    var     temp:integer;
 6    begin
 7      temp:=a;
 8      a:=b;
 9      b:=temp;
10    end;
17  function gcd;
18    var     w:integer;
19    begin
20      if m < n then begin
21        swap(m,n);
22      end;
23      while n <> 0 do begin
24        w:=m mod n;
25        m:=n;
26        n:=w;
27      end;
28      gcd:=m;
29    end;
30  begin
32    readln(x,y);
34    g:=gcd(x,y);
36    writeln('gcd=',g);
38  end.
```

図 3: 元のプログラムとその Backward Slice の比較

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;
 4  procedure swap(var a,b:integer);
 5    var     temp:integer;
 6    begin
 7      temp:=a;
 8      a:=b;
 9      b:=temp;
10    end;
11  function lcm(a,b:integer):integer;
12    var     c:integer;
13    begin
14      c:=gcd(a,b);
15      lcm:=(a div c)*(b div c)*c
16    end;
17  function gcd;
18    var     w:integer;
19    begin
20      if m < n then begin
21        swap(m,n);
22      end;
23      while n <> 0 do begin
24        w:=m mod n;
25        m:=n;
26        n:=w;
27      end;
28      gcd:=m;
29    end;
30  begin
31    writeln('Input x and y');
32    readln(x,y);
33    writeln('x=',x,' y=',y);
34    g:=gcd(x,y);
35    l:=lcm(x,y);
36    writeln('gcd=',g);
37    writeln('lcm=',l);
38  end.
```

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;
11  function lcm(a,b:integer):integer;
12    var     c:integer;
13    begin
14      c:=gcd(a,b);
15      lcm:=(a div c)*(b div c)*c
16    end;
17  function gcd;
18    var     w:integer;
19    begin
23      while n<>0 do begin
24        w:=m mod n;
25        m:=n;
26        n:=w;
27      end;
28      gcd:=m;
29    end;
30  begin
34    g:=gcd(x,y);
35    l:=lcm(x,y);
36    writeln('gcd=',g);
37    writeln('lcm=',l);
38  end.
```

図 4: 元のプログラムとその Forward Slice の比較

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 =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 Ins(p) で表し，実行時点 p に対して，Def(p) は実行時点 p における文 Ins(p) の実行により定義された変数，Use(p) は実行時点 p における文 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 DU(p, q) があるとは，ある変数 $w \in$ Use(q) が存在して，p =Def(q, w) の場合をいう．データ依存関係 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 $w \in$ 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，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 CtlExec(t) を次のように定義する．

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

   ただし，命令 t がループ命令の場合には命令 t も 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 TC(p, q) があるとは，

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

   の場合をいう．
   制御依存関係 TC(p, q) は，実行時点 q において命令 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 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 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 TC(p, q) があるという．このとき実行時点 q における命令 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シング基準といい $C = (x, r, V)$ 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 のスライシング基準 $C = (x, r, V)$ 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 | $VarREF(p) \cup$ Contorol(p)} $\cup$ CalcDsObj
   DS $\leftarrow$ DS $\cup$ {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 のスライシング基準 $C = (x, r, V)$ に関する動的スライスとなる．

2.2.2 動的スライスの計算

ここで実際に 図 5 のプログラムに異なる入力を与え動的スライスを計算する．このプログラムは，５個の数字を読み込みその中の最大値及び最小値を出力するプログラムである．

- 入力に a[1]=1，a[2]=2，a[3]=4，a[4]=3，a[5]=5 を与えて実行したときの実行系列を 図 7 に示す．上記の手順にしたがって 図 7 の実行時点 43 の変数 min に関する動的スライスを計算すると 図 5 の左の動的スライスが得られる．

- 入力に a[1]=5，a[2]=4，a[3]=2，a[4]=3，a[5]=1 を与えて実行したときの実行系列を 図 8 に示す．上記の手順にしたがって 図 7 の実行時点 43 の変数 min に関する動的スライスを計算すると 図 5 の右の動的スライスが得られる．

```
1  program max_min(input,output);
2  var i,max,min:integer;
3      a:array[1..5] of integer;
4  begin
5   i:=1;
6   writeln('Input 5 numbers');
7   while i<=5 do
8   begin
9     readln(a[i]);
10    i:=i+1
11  end;
12  min:=a[1];
13  max:=a[1];
14  i:=2;
15  while i<=5 do
16  begin
17    if min>a[i] then
18       min:=a[i] ;
19    if max<a[i] then
20       max:=a[i] ;
21    i:=i+1
22  end;
23  writeln('MAX=',max);
24  writeln('MIN=',min)
25  end.
```

図 5: プログラム

```
1  program max_min(input,output);
2  var i,max,min:integer;
3      a:array[1..5] of integer;
4  begin
5   i:=1;

7   while i<=5 do
8   begin
9     readln(a[i]);

11  end;
12  min:=a[1];

24  writeln('MIN=',min)
25 end.
```

```
1 program max_min(input,output);
2 var i,max,min:integer;
3     a:array[1..5] of integer;
4 begin
5   i:=1;

7   while i<=5 do
8   begin
9     readln(a[i]);
10    i:=i+1
11  end;
12  min:=a[1];

14  i:=2;
15  while i<=5 do
16  begin
17    if min>a[i] then
18       min:=a[i] ;


21    i:=i+1
22  end;

24  writeln('MIN=',min)
25 end.
```

図 6: 異なる入力による動的スライス

```
 1    i:=1                              {i=1}
 2   writeln('Input 5 numbers')
 3   while i<=5 do                     {1<=5}
 4     readln(a[i])                    {a[1]=1}
 5     i:=i+1                          {i=2}
 6   while i<=5 do                     {2<=5}
 7     readln(a[i])                    {a[2]=2}
 8     i:=i+1                          {i=3}
 9   while i<=5 do                     {3<=5}
10     readln(a[i])                    {a[3]=4}
11     i:=i+1                          {i=4}
12   while i<=5 do                     {4<=5}
13     readln(a[i])                    {a[4]=3}
14     i:=i+1                          {i=5}
15   while i<=5 do                     {5<=5}
16     readln(a[i])                    {a[5]=5}
17     i:=i+1                          {i=6}
18   while i<=5 do                     {6<=5}
19   min:=a[1]                         {min=1}
20   max:=a[1]                         {max=1}
21   i:=2                              {i=2}
22   while i<=5 do                     {5<=5}
23     if min>a[i] then                {1>2}
24     if max<a[i] then                {1<2}
25       max:=a[i]                     {max=2}
26     i:=i+1                          {i=3}
27   while i<=5 do                     {3<=5}
28     if min>a[i] then                {1>4}
29     if max<a[i] then                {2<4}
30       max:=a[i]                     {max=4}
31     i:=i+1                          {i=4}
32   while i<=5 do                     {4<=5}
33     if min>a[i] then                {1>3}
34     if max<a[i] then                {4<3}
35     i:=i+1                          {i=5}
36   while i<=5 do                     {5<=5}
37     if min>a[i] then                {1>5}
38     if max<a[i] then                {4<5}
39       max:=a[i]                     {max=5}
40     i:=i+1                          {i=6}
41   while i<=5 do                     {6<=5}
42   writeln('MAX=',max)               {writeln(5)}
43   writeln('MIN=',min)               {writeln(1)}
```

図 7: 実行系列

```
 1    i:=1                              {i=1}
 2   writeln('Input 5 numbers')
 3   while i<=5 do                     {1<=5}
 4     readln(a[i])                    {a[1]=5}
 5     i:=i+1                          {i=2}
 6   while i<=5 do                     {2<=5}
 7     readln(a[i])                    {a[2]=4}
 8     i:=i+1                          {i=3}
 9   while i<=5 do                     {3<=5}
10     readln(a[i])                    {a[3]=2}
11     i:=i+1                          {i=4}
12   while i<=5 do                     {4<=5}
13     readln(a[i])                    {a[4]=3}
14     i:=i+1                          {i=5}
15   while i<=5 do                     {5<=5}
16     readln(a[i])                    {a[5]=1}
17     i:=i+1                          {i=6}
18   while i<=5 do                     {6<=5}
19   min:=a[1]                         {min=5}
20   max:=a[1]                         {max=5}
21   i:=2                              {i=2}
22   while i<=5 do                     {2<=5}
23     if min>a[i] then                {5>4}
24        min:=a[i]                     {min=4}
25     if max<a[i] then                {5<4}
26     i:=i+1                          {i=3}
27   while i<=5 do                     {3<=5}
28     if min>a[i] then                {4>2}
29        min:=a[i]                     {min=2}
30     if max<a[i] then                {5<2}
31     i:=i+1                          {i=4}
32   while i<=5 do                     {4<=5}
33     if min>a[i] then                {2>3}
34     if max<a[i] then                {5<3}
35     i:=i+1                          {i=5}
36   while i<=5 do                     {5<=5}
37     if min>a[i] then                {2>1}
38        min:=a[i]                     {min=1}
39     if max<a[i] then                {5<1}
40     i:=i+1                          {i=6}
41   while i<=5 do                     {6<=5}
42   writeln('MAX=',max)               {writeln(5)}
43   writeln('MIN=',min)               {writeln(1)}
```

図 8: 実行系列 2

3　静的情報と動的情報を用いたスライス計算法

静的スライスの計算は，プログラムを実行せずソースプログラムの依存解析を行い，得られた PDG 上で行う．PDG を構築する際，制御依存関係はプログラム構造を調べることで容易に求まる．また，データ依存関係はデータフロー方程式 [2] を解くことで求まる．静的スライス計算の複雑さは，その評価の基準によって変わるが，現実には比較的短い時間で計算できる [18, 15]．しかし，すべての実行可能なパスについて考えているため，静的スライスに元のプログラムの大部分が含まれてしまい，デバッグやプログラム理解等の効率向上は期待できない場合がある．

一方，動的スライスは特定の入力でプログラムを実行して得られる実行系列から計算するため，その実行に関係のない部分はスライスに含まれない．つまり，一般的に抽出されるサイズは静的スライスよりも小さくなる．

特定のテストデータに対して不具合が発生し，その欠陥の原因をプログラム中で捜すことを考える．静的スライスでは，そのテストデータでは実行されていないパスに関する依存関係まで計算してしまうため，不具合の原因とは全く関係のない部分までスライスに含まれてしまうが，動的スライスはテストデータの実行に関連する部分の中から計算するため，不具合に関係のない部分がスライスに含まれるようなことはない．

このように特定のテストデータで存在するフォールト位置特定を行う場合には，動的スライスは非常に有効となる．

動的スライスの計算には，プログラム実行前の解析は必要ではないが，実行中に動的データ依存関係と動的制御依存関係の情報を主記憶等に記憶しなければならない．このため，動的スライスの計算には多くの記憶領域と計算時間を要する．また，動的スライスを抽出する対象となる実行系列はプログラムが実行した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くなるために，抽出する時間も非常に長くことがある．

4 Statement-Mark スライス

本節では，文の実行履歴を用いた Statement-Mark スライスを提案する．

4.1 Call-Mark スライス

我々は，静的スライスと動的スライスの中間に位置付けされるものとして，Call-Mark スライス [11, 14] を提案している．Call-Mark スライスの計算手順は，次のようになる．

1. 静的スライスと同じ方法で PDG を作成する．
2. プログラムに入力データを与えて実行させる．その際，関数の実行履歴 (各関数呼び出し文が実行されたかどうか) を取得する．
3. 取得した関数の実行履歴を用いて実行されなかった文の一部を特定し，それらの文をスライスから取り除く．

関数の実行履歴という簡単な動的情報を用いることで，動的スライスの欠点であった実行時のオーバーヘッドを大幅に減らしている．

4.2 Statement-Mark スライスの概要

Call-Mark スライスでは，動的情報として関数の実行履歴を用いているが，これを文の実行履歴にすれば，取得すべき情報は多くなるが，より正確なスライスを抽出することができると考えられる．そこで，文の実行履歴を用いた Statement-Mark スライスを提案する．

4.2.1 アルゴリズム

基本的には Call-Mark スライスと同じ手順で計算を行うことになる．

1. 静的な依存関係解析を行い，PDG を構築する．
2. 入力データを与えて実行する．ある文が実行されるごとに，対応する PDG の節点に対してフラグを立てる．
3. PDG において，スライシング基準に対応する節点から依存辺を逆向きに探索する．実行されていない節点に着いた時は，(その節点も含めて) その節点以降の探索を止め，他の分岐を探す．

4.2.2 例

図 9 のプログラムについてのスライシング基準が入力 $(a = 2, b = 3, c = 0)$，文 24 の変数 d である Statement-Mark スライスの計算結果を 図 10 に示す．

動的スライスと全く同じ結果が得られているが，一般に，動的スライスは Statement-Mark スライスの部分集合である．

4.2.3 評価

Statement-Mark スライスを，実際にスライシングツール [15] に実装し，スライスサイズと実行時間を計測した．(表 1 ， 表 2) スライスサイズは，各々のプログラムに対して 3 つのスライス基準を無作為に選び，得られたスライスの行数を平均したものである．

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7  function Cube(x : integer):integer;
 8  begin
 9     Cube := x*x*x
10  end;
11  begin
12    writeln("Squared Value ?");
13    readln(a);
14    writeln("Cubed Value ?");
15    readln(b);
16    writeln("Select Feature!  Square:0
                                Cube: 1");
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20    else
21       d := Cube(b);
22    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

図 9: ソースプログラム

```
 1  program Square_Cube(input,output);
 2  var a,b,c,d : integer;
 3  function Square(x : integer):integer;
 4  begin
 5     Square := x*x
 6  end;
 7
 8
 9
10
11  begin
12
13    readln(a);
14
15
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20
21
22
23
24    writeln(d)
25  end.
```

図 10: 入力データ $(a = 2, b = 3, c = 0)$,24 行目変数 d に関する Statement-Mark スライス

表 1: スライスサイズ (行)

program	static	Call-Mark	Statement -Mark	dynamic
P1	20.7	16.7	16.0	4.7
P2	182	162	143	5.3
P3	187	166	149	7.7

(Celeron-450MHz with 128MB Memory)

表 2: 実行時間 (ms)

program	static	Call-Mark	Statement -Mark	dynamic
P1	52	53	53	184
P2	46	46	47	4,649
P3	4,869	4,899	4,955	38,969

(Celeron-450MHz with 128MB Memory)

5 部分解析法

静的スライスにおいては，入力データを考慮せずに全ての可能な実行パスを想定して依存関係解析を行っている．しかし 1 度入力データを与えて実行すれば，その実行において使用されていない文が特定できる．使用されなかった部分を解析しないことによって，

1. スライスのサイズの減少
2. PDG 構築のコストの削減

が期待できる．

5.1 部分解析法の概要

Call-Mark スライス計算法を改良した，部分解析法を提案する．

ある実行 E において使用された関数の集合を $UsedF(E)$ とする．[18] のアルゴリズムは，

1. 関数内の依存関係解析
2. プログラム全体の依存関係解析

となっている．そこで，$f \notin UsedF(E)$ である関数 f の依存解析は行わなければよい．また，プログラム全体の解析の際には，関数呼び出し文の出現による PDG の接続等の作業を行わない．

この部分解析法は，関数の実行履歴を利用している．このため，実行時間，スライスサイズは Call-Mark スライスと等しいスライスが得られる．また，文の実行履歴を利用することで，さらに効率良い依存関係解析が期待できる．その際に得られるスライスは，Statement-Mark スライスとほぼ同等のものとなる．

5.1.1 評価

Statement-Mark スライスを，実際にスライシングツール [15] に実装し，解析時間を計測した．(表 3)

結果として，静的スライスと比較すると，解析時間に関して 30~50%の減少が見られた．

5.1.2 適用例

部分解析法は，何らかの形で実行されていない文を調べ，その部分を解析しないことによって，そのコストを削減するというものである．

そこで，関数の実行履歴を用いることで，ソースプログラムの何% が不要になるかを調べた．

対象となるプログラムとして，ftp を選んだ．このプログラムは，ソースプログラムが 6294 行で，定義された関数が 130 個である．ftp を実際に実行し，使われなかった関数をソースプログラムから削除し，解析が必要な部分がどれだけかを調べた．

2 回の実行を行い，データを取得した．

1 回目は簡単な実行で，ls でファイルを確かめ，get でファイルを獲得し，bye で終了するというものであった．この実行では，30 個の関数が使用され，必要な部分は 3227 行 (51.3%) であることが分かった．

2 回目は 73 個のコマンドを実行するものであった．この実行では，87 個の関数が使用され，必要な部分は 4838 行 (76.9%) であると分かった．

つまり，一般的なプログラムにおいても解析が必要な部分はかなり限られてくるため，この手法は有効であるということがいえる．

5.2 Call-Mark スライス，Statement-Mark スライス，部分解析法の特徴

関数の実行履歴を用いた部分解析法と Call-Mark スライス，文の実行履歴を用いた部分解析法と Statement-Mark スライスは，共にスライスのサイズとしてはほぼ同等になる．

部分解析法を用いれば，他の 2 つのスライス技法と比べて，依存関係解析のコストを減らすことができる．しかし，入力データに依存した PDG を構築するため，別の実行を行った際には，新たに PDG を再構築する必要がある．

再構築を避けるためには，再実行した際に PDG を完全に作り直すのではなく，incremental に必要な部分のみを解析し，PDG を作るという方法も考えられる．

表 3: 解析時間 (ms)

program	static	Call-Mark	partial
P1	14	15	8.7
P2	219	230	156
P3	710	754	346

(Celeron-450MHz with 128MB Memory)

6 D3 スライス

6.1 配列，ポインタの解析

配列変数を静的に解析する際に，配列の添字の値を把握するのは困難であり，それゆえ不要に多くの依存関係を抽出してしまうことがある．

図 11 は，配列を含んだ簡単なプログラムである．文 6 の変数 c の値が分からないため，文 6 の配列 a が文 1~3 全てにデータ依存していると解析される．

ポインタ解析では，ポインタによる alias によって，陽には表れないデータ依存関係が発生する．その依存関係を知るために，各ポインタが何を指しているかを把握しておく必要がある．point-to グラフを使った手法 [17] などが提案されているが，多大なコストを必要とするものが多く，静的に解析する方法には限界がある．

図 12 は，ポインタを利用する簡単なプログラムである．文 7 の変数 a は文 6 によって定義されているが，こういった依存関係は簡単には解析できない．

6.2 方針

配列，ポインタの静的な解析は，コストがかかる上に完全な解析は期待できない．そこで，実行中にデータ依存関係を抽出する方法が考えられる．実行中は，配列の添字，ポインタの参照先を把握するのは簡単である．動的スライスとの差異として，次の 2 つが挙げられる．

- 制御依存関係は静的に解析する．
- 実行系列を保存することはしない．

これによって，動的スライス抽出に比べて実行時間の短縮を図る．

6.3 概要

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．

6.4 アルゴリズム

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．

(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．

(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．

6.5 例

図 11 のプログラムに対して，本アルゴリズムを適用した例を示す．

各実行時点における $DefS$ の推移を 表 4 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 $DefS(a[0]) = 1, DefS(a[1]) = 2, DefS([2]) = 3$ となる．

文 4 では，c を定義しているので，$DefS(c) = 4$ となる．

文 5 では，$a[0], c$ を参照しているので，$DDS(5) \leftarrow DDS(5) \cup (a[0], DefS(a[0])) \cup (c, DefS(c))$．つまり，$DDS(5) = \{(a[0], 1), (c, 4)\}$ となる．

図 12 のプログラムに対して，本アルゴリズムを適用した例を示す．

各実行時点における $DefS$ の推移を 表 5 に表す．

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 $DefS(a) = 1, DefS(b) = 2, DefS(c) = 3, DefS(d) = 4$ となる．

[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．

文 5 では，c を参照しているので，$DDS(5) \rightarrow \phi \cup (c, DefS(c))$．つまり，$DDS(5) = (c, 3)$ となる．また，$*c$ を定義しているので，$DefS(*c) = 5 (DefS(a) = 5)$ となる．

文 6 では，b，d，$*d$ を参照しているので，$DDS(6) = \{(b, 2), (d, 4), (*d, 3)\}$ となる．

さらに，文 6 では $**d$ を定義しているので，$Def(**d) = 6 (DefS(a) = 6)$ となる．

文 7 では，a を参照しているので，$DDS(7) = \{(a, 6)\}$ となる．

文 8 では，$d, *d, **d$ を参照しているので，$DDS(8) = \{(d, 4), (*d, 3), (**d, 6)\}$ となる．

6.6 評価

本手法の実現方法は 2 種類が考えられる．

- スライシングツールのインタプリタ部に実装する．
- コンパイラの前処理部として実装する．

今回はこの 2 種類の方法で本手法を評価した．

6.6.1 インタプリタ

本手法を，実際にスライシングツール [15] に実装し，スライスサイズ，解析時間，実行時間を計測した．(表 6 ， 表 7 ， 表 8)

このスライシングツールの対象言語はポインタや構造体を持たない Pascal のサブセットとなっていたが，今回の実装にあたってそれらをサポートすることにした．

スライスサイズを比較すると，本手法は動的スライスにかなり近いサイズになっていることが分かる．また，解析時間を比較すると，静的スライスの 7~35%の解析時間となっており，制御依存関係解析にそれほど時間がかかっていないことが分かる．実行時間を比較すると，本手法は静的スライスとほとんど同じ値となっている (5%程度の差)．

P4 のプログラムに関しては，スライスサイズと解析時間が N/A になっているが，これはダイナミックスライサがポインタ拡張されておらず，データがとれなかったためである．

6.6.2 コンパイラ

実際にコンパイラの前処理部分をつくる時間が不足していたため，実際のプログラムと，自分で前処理を書き足したプログラムとで，実行時間の差異を調べた．

サンプルプログラムとして，1000 個の要素に対してマージソートを行うプログラムを用いた．元のプログラムの実行時間は 2,134ms かかったのに対し，データ依存解析を行うプログラムは 15,567ms かかっており，約 7.3 倍の時間がかかることがわかった．

6.7 考察

我々のスライシングツールの実行部はインタプリタとなっているため，1 文の実行にかかる時間が大きい．このため，相対的に依存関係解析のコストが少ないものになってしまう．

しかし 6.6.2 からわかるように，コンパイラの場合は 1 文あたりの実行にかかる時間が短いため，もとのプログラムの数倍程度の実行時間が必要となる．

そこで，この手法をさらに簡易化した手法が考えられる．元々本手法は，配列やポインタを詳しく解析するための手法であった．そこで，配列，ポインタに関するデータ依存関係解析を動的に行い，その他一般の変数のデータ依存関係と制御依存関係を静的に行う方法が考えられる．

この限定手法に関して， 6.6.2 と同じようにデータを取ったところ，3.5 倍程度の時間がかかることが判明し，元の手法の約半分程度の時間での解析が期待できる．

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

図 11: 配列を含むプログラム

```
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 12: ポインタを利用するプログラム

表 4: 図 11 のプログラムにおける，各実行時点での $DefS$ の推移

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

表 5: 図 12 のプログラムにおける，各実行時点での $DefS$ の推移

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

表 6: スライスサイズ (行)

program	static	D3	dynamic
P1	20.7	15.0	4.7
P2	182	15.7	5.3
P3	187	61	7.7
P4	60	13	N/A

(Celeron-450MHz with 128MB Memory)

表 7: 解析時間 (ms)

program	static	D3	dynamic
P1	14	4.5	N/A
P2	219	19	N/A
P3	710	48	N/A
P4	32	11	N/A

(Celeron-450MHz with 128MB Memory)

表 8: 実行時間 (ms)

program	static	D3	dynamic
P1	52	55	184
P2	46	49	4,649
P3	4,869	5,274	38,969
P4	210	226	N/A

(Celeron-450MHz with 128MB Memory)

7 考察

各手法の解析に必要なメモリ量について考察する．

ソースプログラムの文の数を s，定義された変数の数を v，ソースプログラム中の実行された関数の割合を $\mu(\mu \leq 1)$，実行系列の長さを e とする．一般的には $s \ll e$ となることが多い．

静的スライスでは PDG を構築する．PDG の節点数は s である．制御依存辺の数が最大 s^2 であり，データ依存辺の数が最大 vs^2 であるため，必要メモリ量は $O((v+1)s^2)$ となる．

動的スライスでは DDG を構築する．DDG の節点数は e であり，データ依存辺の種類が v 個存在する．よって，最大 ve^2 のデータ依存辺が引かれる可能性があるため，必要メモリ量は $O(ve^2)$ となる．

Call-Mark スライスでは PDG を構築する．このため，PDG の構築に必要なコストは静的スライスと同じである．この手法ではさらに実行時に各文にフラグを立てる．この際に必要なメモリ量は $O(\mu s)$ である．このため，必要メモリ量は $O((v+1)s^2)$ となる．

Statement-Mark スライスでは PDG を構築する．このため，PDG の構築に必要なコストは静的スライスと同じである．この手法ではさらに実行時に各文にフラグを立てる．この際に必要なメモリ量は s である．このため，必要メモリ量は $O((v+1)s^2)$ となる．

部分解析法におけるグラフの節点数は μs である．また実行時に必要なメモリ量は μs である．このため，必要メモリ量は $O((v+1)(\mu s)^2)$ となる．

D3 スライスにおけるグラフの節点数は s である．また実行時に必要なメモリ量は，実行中に利用される変数の数 (配列や構造体の各要素，動的に生成される変数も 1 つと数える) に比例する．これは最大 e である．また，データ依存辺の種類が v 個存在するので，データ依存辺は最大 vs^2 個引かれる可能性がある．このため，必要メモリ量は $O(vs^2)$ となる．

8 まとめ

本研究では，静的情報と動的情報を組み合わせ，正確性 (スライスサイズ) と必要コストを考慮に入れた 3 つのスライス抽出アルゴリズムを提案した．

1. Statement-Mark スライス
2. 部分解析法
3. 動的なデータ依存関係解析法

1. は，Call-Mark スライスよりも詳細な動的情報 (文の実行履歴) を用いることで，より正確性を向上させる手法である．

2. は，依存関係解析の際に動的情報 (関数の実行履歴) を用いることで，正確性の向上だけでなく，解析に必要なコストを減少させる手法である．

3. は，配列やポインタを詳細に解析するために，データ依存関係のみを動的に解析する手法である．

またこれらの手法をツールに実装し，従来手法との比較を行い，本手法の正当性を確認した．

今後の課題としては，

- 大規模プログラムに対して，これらの手法を適用する．
- 作成予定の新しいスライシングツール (対象言語: Java) に本手法を適用する．
- スライス計算対象のプログラムの内容，許容できる計算時間等により，どの手法を用いることが妥当であるかを判定できるようなメトリクスを提案する．

といったことがあげられる．

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 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頂きました同 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頂きました同 松下誠 助手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] Aho, A.V., Sethi, R., and Ullman, J.D. : "Compilers: Principles, Techniques, and Tools", *Addison Wesley*, Massachusetts, 1986.

[3] 芦田, 西松, 楠本, 井上 : "プログラムスライスに基づいたデバッグ支援システムの評価のための追証実験", **第** *58* **回** *(***平成** *11* **年前期***)* **全国大会**, **講演論文集** *(1)*, pp. 259–260, 1999.

[4] 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, 1999.

[5] Bates, S. and Horwitz, S. : "Incremental Program Testing Using Program Dependence Graphs", *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

[6] Beck, J. and Eichmann, D. : "Program and Interface Slicing for Reverse Engineering", *Proceedings of the 15th International Conference on Software Engineering*, pp. 509–518, 1993.

[7] Binkley, D.W. and Gallagher, K.B. : "Program Slicing", *Advances in Computers, Volume 43*, Marvin Zelkowitz, Editor, Academic Press San Diego, CA, 1996.

[8] Gallagher, K.B. and Lyle, J.R. : "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering, 17(8)*, pp. 751–761, 1991.

[9] Gupta, R., Soffa, M.L., and Howard, J. : "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *ACM Transaction on Software Engineering and Methodology, Vol. 6, No. 4*, pp. 370–397, 1997.

[10] 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.

[11] 地平, 西松, 楠本, 井上: "関数呼び出し履歴を利用したプログラムスライスの提案と実現", **電子情報通信学会技術研究報告**, *SS98-7*, pp. 9–15, 1998.

[12] 西江, 神谷, 楠本, 井上: "プログラムスライスに基づくデバッグ支援ツールの実験的評価", **ソフトウェアシンポジウム** *97* **予稿集**, pp. 142–147, 1997.

[13] 西松, 楠本, 井上: "フォールト位置特定におけるプログラムスライスの実験的評価", **電子情報通信学会技術研究報告**, *SS98-3*, pp. 17–24, 1998.

[14] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K. : "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", *Proceedings of The 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999.

[15] 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.

[16] 下村 隆夫: "Program Slicing 技術とテスト，デバッグ，保守への応用", **情報処理**, *Vol. 33, No. 9*, pp. 1078–1086, 1992.

[17] Steensgaard, B. : "Points-to analysis in almost linear time", *Technical Report MSR-TR-95-08*, Microsoft Research, 1995.

[18] 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.

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