修士学位論文

題目

プログラムスライス抽出技法の効率制御のための一手法

指導教官
井上克郎 教授

報告者
西松 顯

平成 11 年 2 月 16 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 10 年度 修士学位論文

プログラムスライス抽出技法の効率制御のための一手法

西松 顯

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．しかし，大規模なプログラムに対するスライス抽出には非常に時間を要することが指摘されており [1]，これまでにスライス抽出に要する時間を削減するための手法が提案されている [1, 2]．

スライスを抽出するためには，プログラムの文間の依存解析を行い，プログラム依存グラフ (Program Dependence Graph) を構築する必要がある．従来の手法では，スライスを抽出するプログラムにより，簡略化された依存解析アルゴリズムを用いることで，時間的問題を解決していた．しかし，従来の手法では依存解析アルゴリズム間で依存関係情報を共有，再利用していないために，ある依存解析アルゴリズムが生成した依存関係情報が存在している場合でも，他のアルゴリズムにより，その依存関係情報は利用できないために，新たに依存解析を最初から行う必要があった．

本研究では，3 種類の簡略化の度合が異なる依存解析アルゴリズム間で依存関係情報を共有，再利用可能にすることで，精度の異なるスライスを効率良く抽出するための手法を提案する．また，本手法をツール [17] に実装し，従来手法との比較を行い，有効性を確認する．

主な用語

プログラムスライス (Program Slice)
プログラム依存グラフ (Program Dependence Graph)
効率 (Efficiency)
精度 (Correctness)

目 次

1 まえがき **3**

2 プログラムスライス **5**

2.1 スライス計算手順 · · · · · 5

2.1.1 Phase 1:依存関係解析 · · · · · 6

2.1.2 Phase 2:スライス抽出 · · · · · 9

2.2 従来手法の問題点 · · · · · 11

2.3 研究の適用範囲 · · · · · 14

3 CIFIグラフ構築アルゴリズム **17**

3.1 プログラム表現 · · · · · 17

3.2 関数内アルゴリズム · · · · · 19

3.3 関数間アルゴリズム · · · · · 20

4 CIFSグラフ構築アルゴリズム **23**

4.1 方針 · · · · · 23

4.2 データ依存関係の計算方法 · · · · · 24

4.2.1 諸定義 · · · · · 24

4.2.2 逆方向のデータ依存関係の計算方法 · · · · · 26

4.2.3 順方向のデータ依存関係の計算方法 · · · · · 27

4.3 アルゴリズム · · · · · 28

5 実行効率 **30**

5.1 計算量 · · · · · 30

5.1.1 CIFI-PDGを作るコスト · · · · · 30

5.1.2 CIFI-PDGを用いてスライスを計算するコスト · · · · · 31

5.2 実装 · · · · · 31

5.3 考察 · · · · · 33

6 まとめ **35**

謝辞 **36**

参考文献 **37**

1 まえがき

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 およびある変数 v に対して，n における v の値に影響を与える P 中の各文や式の集合を言う．すべての可能な入力データに対して解析したものを静的スライス，特定の入力データに対して解析したものを動的スライスという (本論文では静的スライスのみを扱うので，以下単にスライスと言えば，静的スライスを意味するものとする)．スライスの抽出技法は Mark Weiser [21] によって提案され, 当初はプログラムのデバッグを支援するために使われていたが，現在では，デバッグだけでなくテストや保守，プログラム合成などにも利用されている [5, 18]．

当初，スライス計算の対象となるプログラム言語は，非常に単純な言語であったが，実用的な言語 (例えば，C 言語) にスライスを適用すると，さまざまな問題点が指摘されるようになった．その問題点とは『スライスに含まれるべきでない文がスライスに含まれる』『スライスに含まれるべき文がスライスに含まれない』の 2 点である．例えば，構造化されたプログラム言語では，Weiser が提案している手法により依存解析が可能であるが，非構造な文 (break，continue 等) を含むようなプログラムに対しては，Weiser が提案している手法では，正確な依存解析が行えない．正確な依存解析が行えない場合には，スライスを用いてフォールト位置特定を行う際に，スライス内にフォールトを含まない可能性がある．そこで，これまでに依存関係の正確性を向上させる事を目的とする研究が多数されている (Calling Context[5]，関数間制御依存関係 [4]，statement side-effect[8, 9] 等)．

依存関係の正確性が向上するにつれ，スライス計算アルゴリズムも複雑になり，大規模なソフトウェアに対するスライス計算には多くの時間を要するようになった．例えば，約 28000 行のプログラムに対してスライスを計算するために，約 2 時間を要するという報告がされている [25]．文献 [1] では，スライスの正確性とスライス計算に要する時間とのトレードオフを考慮する必要性が指摘されている．この問題に対して Atkison らは，ユーザがスライスの正確性を制御できるようなシステムを提供する事で，解決できる事を示した [1]．さらに，Tonella らは 3 種類の正確性の異なる依存解析アルゴリズムを有するシステムを開発し，スライス計算の対象プログラムの規模や，得られるスライスの利用方法により適切な依存解析アルゴリズムを利用することで，時間的問題を解決する方法を提案している [2]．しかし，これらの研究においては，各依存解析アルゴリズムにより得られた依存解析情報を共有していないために，依存解析情報の再利用ができず，それぞれの依存解析アルゴリズムに対して同様の解析を何度も行う必要がある．

本研究では，複数の簡略化の異なる依存解析アルゴリズムの生成する依存関係情報を，各依存解析アルゴリズム間で共有，再利用し，精度の異なるプログラムスライスを効率良く抽出するための手法を提案する．具体的には，まず各依存解析アルゴリズム間で共有可能な

依存情報のデータ構造を定義し，次に共有可能なデータ構造を持つ依存情報の生成方法を提案し，最後に各依存解析アルゴリズム間での依存情報の再利用の方法を提案する．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，本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 P のスライスとは，直感的には P 中のある地点 n およびある変数 v に対して，n における v の値に影響を与える P 中の各文や式の集合を言う．スライス抽出技法は Mark Weiser [21] によって提案され, 当初はプログラムのデバッグを支援するために使われていたが，現在では，デバッグだけでなくテストや保守，プログラム合成などにも利用されている [5, 18]．

2.1 スライス計算手順

これまでに我々は文献 [20] において，スライス計算アルゴリズムを提案している．ここでは，文献 [20] のスライス計算手順について簡単に述べる．スライス計算手順は 図 1 に示す 2 つのフェーズ (Phase 1:依存関係解析，Phase 2:スライス抽出) に分けられる．以降，2 つのフェーズについて述べる．

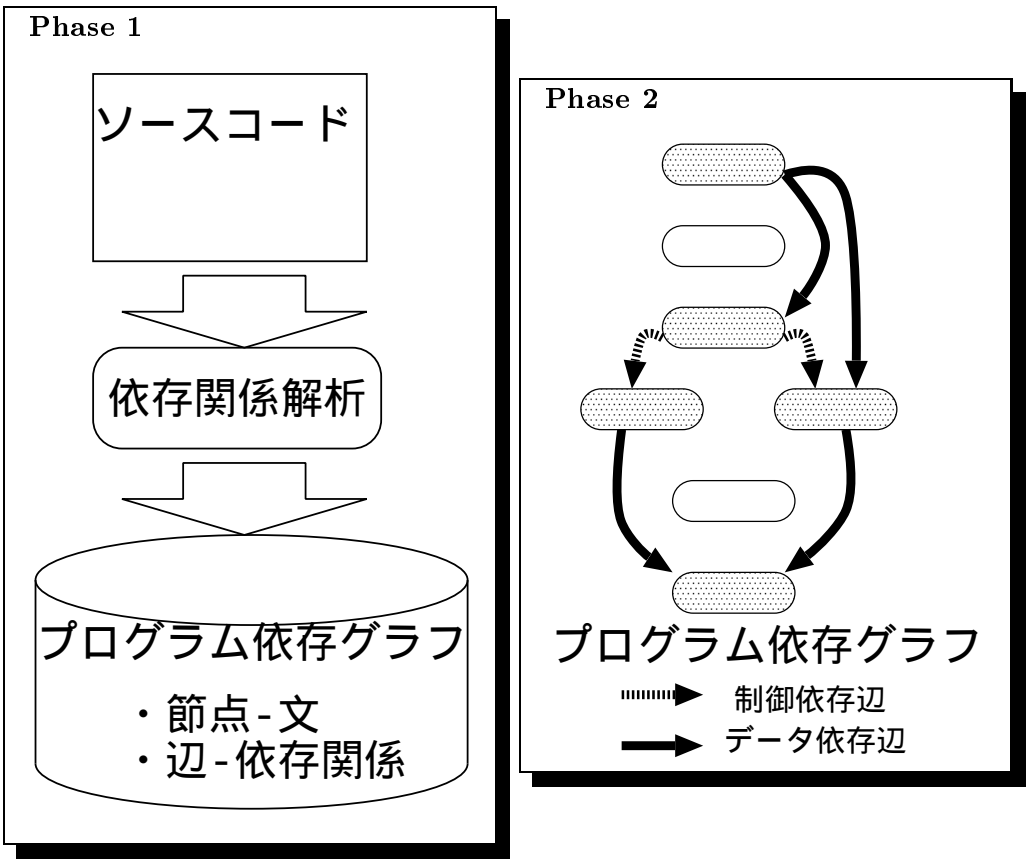

図 1: スライス計算手順

2.1.1 Phase 1:依存関係解析

フェーズ 1 では，まずソースコードを入力とし，プログラム中の文間の依存関係解析を行う (詳細は，文献 [20] を参照)．依存関係には以下の制御依存関係とデータ依存関係の 2 種類がある．

- 制御依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の条件を全て満たしているとき，文 s_1 から文 s_2 への制御依存 *(Control Dependence，CD)* があるという．
  - 文 s_1 が条件文である．
  - 文 s_2 が実行されるかどうかは，文 s_1 の結果に依存する．

  この関係を $s_1 \dashrightarrow s_2$ と表す．

- データ依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の 3 つの条件を全て満たすとき，文 s_1 から文 s_2 へ変数 v に関してデータ依存 *(Data Dependence，DD)* があるという．
  - 文 s_1 で変数 v を定義している．
  - 文 s_2 で変数 v を参照している．
  - 文 s_1 から文 s_2 への実行可能なパスで，そのパスにおいて文 s_1 から文 s_2 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．

  この関係を $s_1 \xrightarrow{v} s_2$ と表す．

次にプログラムに対する依存関係解析で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 と呼ぶ) 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，手続き呼びだし文を表し, その有向辺は 2 つの節点間の制御依存およびデータ依存関係を表す (それぞれ制御依存辺，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 (詳細は文献 [20] を参照)． 図2 のプログラムの PDG を 図3 に示す．

```
 1  program euclid(input,output);
 2    var     x,y,gcdv,lcmv:integer;
 3  function gcd(m,n:integer):integer;forward;
 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    gcdv:=gcd(x,y);
35    lcmv:=lcm(x,y);
36    writeln('gcd=',gcdv);
37    writeln('lcm=',lcmv);
38  end.
```

図 2: プログラム

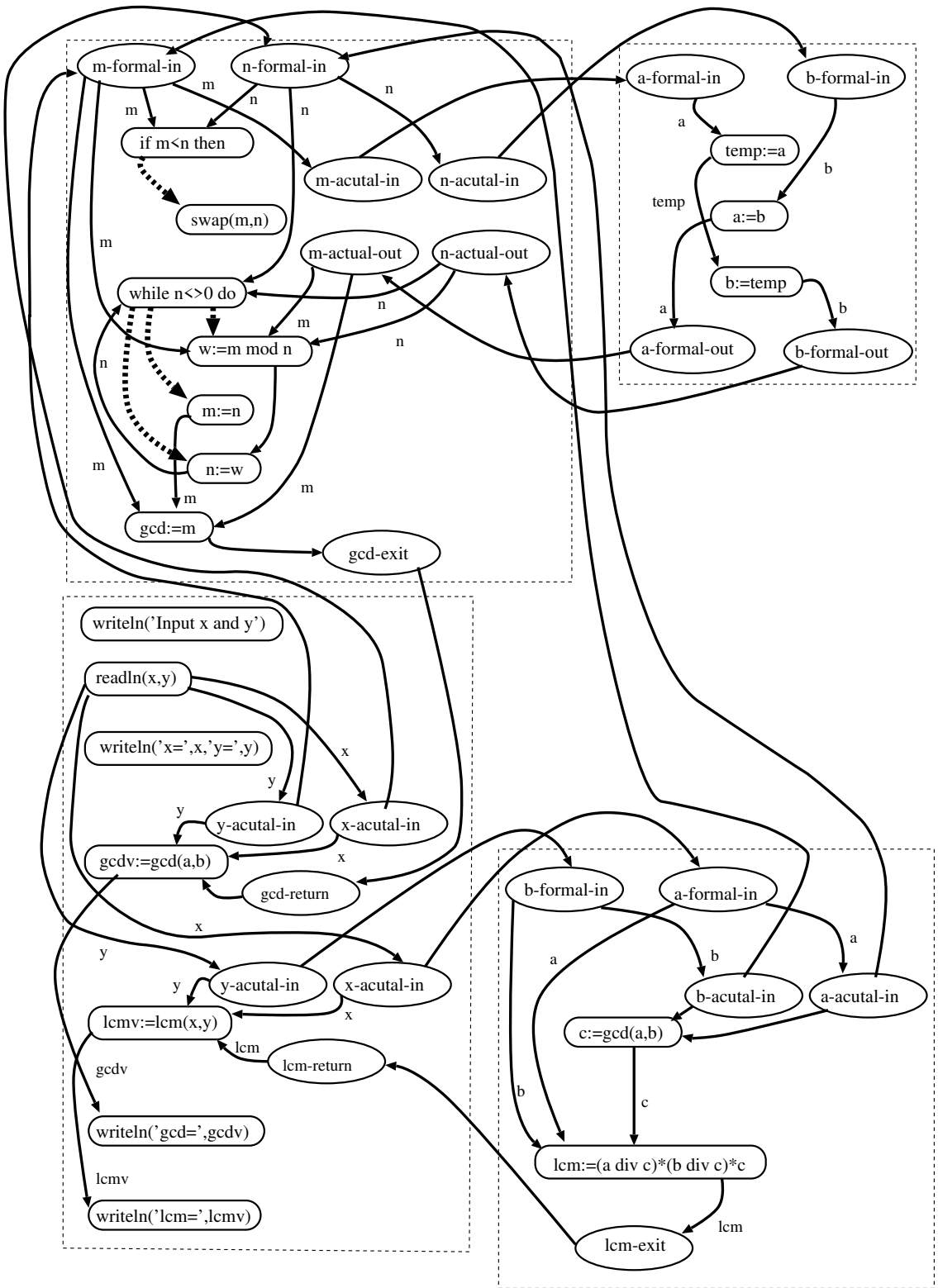
m-formal-in
n-formal-in
if m<n then
swap(m,n)
m-acutal-in
n-acutal-in
m-actual-out
n-actual-out
while n<>0 do
w:=m mod n
m:=n
n:=w
gcd:=m
gcd-exit
a-formal-in
b-formal-in
temp:=a
a:=b
b:=temp
a-formal-out
b-formal-out
writeln('Input x and y')
readln(x,y)
writeln('x=',x,'y=',y)
y-acutal-in
x-acutal-in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

図 3: PDGの例

2.1.2 Phase 2:スライス抽出

フェーズ2では，フェーズ1で生成されたPDGを用いてスライスを計算する．スライスを計算するには『どの文のどの変数に対してスライスを計算するか』を指定する．これをスライシング基準と呼ぶ．スライシング基準は，文と変数の組から成る．スライス計算アルゴリズムを以下に示す．PDG上でのスライスの計算は節点間の到達可能性を判定する問題と同じであるが[16]，PDGには手続きの境界を越えてスライスを計算できるようにいくつかの拡張がなされているので, たどることのできる辺に制限が付く. プログラム内の文 s での変数 v に関するスライスを表す節点の集合 V は, 以下に示すような手順で計算できる. ただし, n_s は s に対応する節点である.

アルゴリズム GetSliceOnPDG

Input: PDG, 文 s , 変数 v

Output: スライスを表す節点の集合 V

Step 1. $V \leftarrow \{n_s\}$

Step 2. $N \leftarrow \{n \mid n \xrightarrow{v} n_s\}$[1] $\cup \{m \mid m \dashrightarrow n_s\}$

Step 3. $N \neq \phi$ の間, 以下の操作を繰り返す

(a) $n \in N$ をひとつ選ぶ

(b) $N \leftarrow N - \{n\}$

(c) $V \leftarrow V \cup \{n\}$

(d) n が in 節点でも引数節点でもないなら次の操作をする

$$N \leftarrow N \cup \{m \mid m \notin V \wedge (m \longrightarrow n \vee m \dashrightarrow n)\}$$

(ただし $m \longrightarrow n$ は m から n への任意の変数のDD関係辺を表す)

図3のPDG上で，スライシング基準 <writeln('gcd=',gcdv) , gvdv> としてアルゴリズムGetSliceOnPDGを実行した結果を 図5 に示す．影が付いている節点がスライスに含まれる節点であり，この節点集合に対応するソースコードの部分を集めたものが 図5 の右のプログラムとなる．

スライス計算において，ソースコードをPDGに変換した後は，単にグラフの到達性を判定する問題として処理できるため，依存関係解析と比較すると，非常に短い時間ですむ．従って，スライス計算における効率とは，主にフェーズ1の依存関係解析の効率のことをいう．

[1] より正確には $\{n \mid \langle v, n\rangle \in RD_{in}(s)\}$ であるが, 一般に, ユーザーは変数 v の参照を含む文を s に指定する場合が多いのでここでは, このわかりやすい表現を用いた.

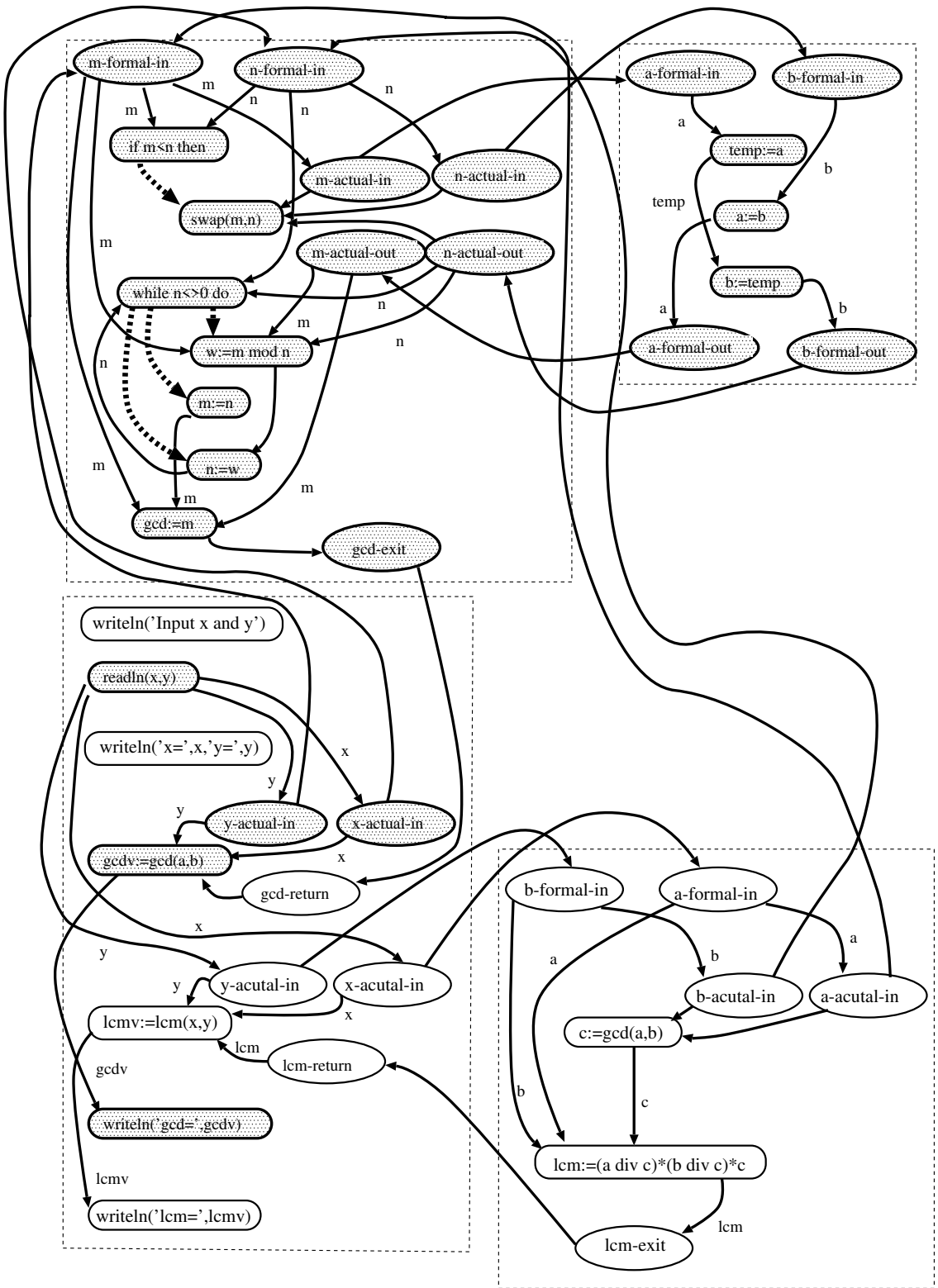

図 4: PDG 上でのスライス計算

```
 1  program euclid(input,output);
 2    var     x,y,gcdv,lcmv:integer;
 3  function gcd(m,n:integer):integer;forward;
 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    gcdv:=gcd(x,y);
35    lcmv:=lcm(x,y);
36    writeln('gcd=',gcdv);
37    writeln('lcm=',lcmv);
38  end.
```

```
 1  program euclid(input,output);
 2    var     x,y,g,l:integer;
 3  function gcd(m,n:integer):integer;forward;
 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 5: 元のプログラムとその Slice の比較

2.2 従来手法の問題点

これまでに，スライス技法を実際に大規模プログラムに適用すると，多くの時間が要する事が指摘されている [1]．例えば，約 28000 行のプログラムに対してスライスを計算するために，約 2 時間を要する [25] という報告がされている．スライスはデバッグにおけるフォールト位置特定作業に有効であるといわれていおり，我々は実際にその有効性を評価を評価している [12]．フォールトを発見した後にフォールト位置特定作業を開始するまでに，スライス計算に多くの時間を要することは非常に問題であり，これまでにスライス計算に要する時間を短縮するために，フェーズ 1 の依存関係解析の効率を向上を目的とする研究がされてき

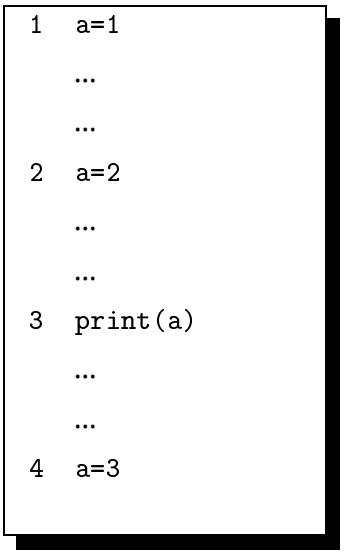

図 6: 制御フロー

```
int a;
main()
{
1   a=1;
2   f(2);
3   print(a);
4   a=3;
5   print(a);
6   f(4);
7   print(a);
}
f(int x)
{
8   if(x)
9       a+=x;
10  return;
}
```

図 7: Calling-Context

た [1, 2]．これらの研究において，依存関係解析の効率向上のために利用されたものとして，(1) 制御フロー，(2)Calling-Context 等がある．

・制御フロー

プログラム中の制御の流れ，つまり，文の実行順序のことである．

・**Calling-Context**

各関数/手続き呼出し文と各関数/手続き本体との対応．

図 6 及び 図 7 を用いて，それぞれ，制御フローを考慮するかどうか，Calling-Context を考慮するかどうかによって得られるデータ依存関係ついて考える．

図 6 の文 3 の変数 a とデータ依存関係である文を調べる．制御フローを考慮にいれる場合には，

- 文 1 の a の定義が文 2 の定義により消去される．

- 文3は文2の定義を参照している．

という情報を得ることができ，結果として，文3の変数aとデータ依存関係がある文は，文2であることが分かる．一方，制御フローを考慮にいれない場合には，

- 変数aが定義されている文の情報を得る．
  $\{1, 2, 4\}$
- 文3のaは$\{1, 2, 4\}$の定義を参照している．

という情報を得ることができ，結果として，文3の変数aとデータ依存関係がある文は，文1，2，9であることが分かる．このように，制御フローを考慮する場合には，正確な情報を得ることができるが，情報を得るための計算量は多い，また，制御フローを考慮しない場合には，情報は正確ではないが，情報を得るための計算量は少なくてすむ．

図7の文3の変数aとデータ依存関係がある文を調べる．Calling-Contextを考慮にいれる場合には，

- 文2の関数に到達した時点で，関数f内を解析する．
- 関数f内の解析後，mainに戻るときに文2に戻る．

という情報を得ることができ，結果として，文3とデータ依存関係がある文は，文1，9であることが分かる．一方，Calling-Contextを考慮にいれない場合には，

- 関数fは，文2，6で呼ばれているという情報を持つ．
- 関数f内の解析後，文2，6の両方に戻る．

という情報を得ることができ，結果として，文3とデータ依存関係がある文は，文1，4，9であることが分かる．Calling-Contextに関しても，制御フローの場合と同様に，情報の正確性と計算量の関係が言える．

制御フロー，Calling-Contextの2つを考慮しているかどうかで，以下の3種類[2]の依存関係解析アルゴリズムが存在する[2]．

1. CSFS(Context Sensitive Flow Sensitive)
2. CIFS(Context Insensitive Flow Sensitive)
3. CIFI(Context Insensitive Flow Insensitive)

[2]Calling Contextを考慮にいれる場合には制御フローも考慮にいれる必要があるためCSFIは存在しない．

これら 3 種類のアルゴリズムの計算量，情報の正確性を比較すると， 表 1 のようになる [2]．文献 [2] では，上記の 3 種類のアルゴリズム (CIFI，CIFS，CSFS) を持つシステムを開発している．しかし，このシステムは 図 8 に示すように，各アルゴリズムにより得られた依存関係情報を共有していないために，アルゴリズム間での依存関係情報の再利用がされていない．例えば，ある文のある変数に関するデータ依存関係を CIFI で計算した後に，さらに CIFS で計算しようとすると，CIFI で得られた依存関係情報を全く用いずに，最初から CIFS で計算を行うために，非常に効率の悪いものとなっている．

表 1: アルゴリズム比較

	CSFS	CIFS	CIFI
計算量	多	中	少
正確性	高	中	低

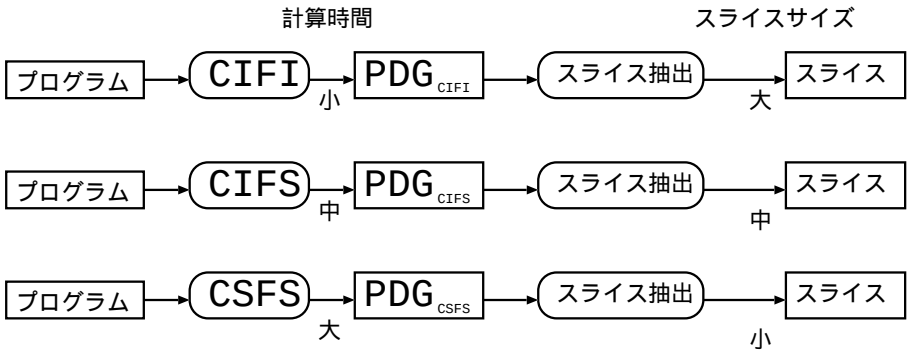

図 8: 従来手法

2.3 研究の適用範囲

本研究では，CIFI，CIFS，CSFS により得られた依存関係情報を共有，再利用することが可能な手法を提案する．本手法により，例えば CIFI により得られた依存関係情報を用いて CIFS を行うことができ，結果として単純に CIFS を行うよりも高速に行え，また，それぞれの解析手法により得られた依存関係情報を別々に管理するのではなく，単一のプログラム表現として扱えるため空間的コストが少なく済む．上記のようなシステムでは，以下の 6 種類の変換が必要となる．[3] ．

[3] PDG_{CIFI} とは，CIFI 依存解析アルゴリズムにより生成される PDG を表す．他も同様である．

1. ソースコード $\Rightarrow$ CIFI $\Rightarrow PDG_{CIFI}$
2. ソースコード $\Rightarrow$ CIFS $\Rightarrow PDG_{CIFS}$
3. ソースコード $\Rightarrow$ CSFS $\Rightarrow PDG_{CSFS}$
4. $PDG_{CIFI} \Rightarrow$ CIFS $\Rightarrow PDG_{CIFS}$
5. $PDG_{CIFI} \Rightarrow$ CSFS $\Rightarrow PDG_{CSFS}$
6. $PDG_{CIFS} \Rightarrow$ CSFS $\Rightarrow PDG_{CSFS}$

しかし，このうち (2)(3) に関しては，それぞれ単独に実装している 図 8 と同様の手法を利用できるために，本研究では述べない．さらに，(5) に関しても，(2) と (6) を連続して行う事と同等であるため述べない．また，(6) に関しては，既に文献 [5] で，実現する手法が提案されているため述べない．よって，ここでは (4) に関して，また，(4) で再利用可能な PDG を生成するための (1) の手法についても述べる (図 9)．以降， 3 節で 図 9 の (1) について述べ， 4 節で (4) について述べる．以降，(1) を『CIFI グラフ構築』，(4) を『CIFS グラフ構築』と呼ぶことにする．

以降で紹介する解析アルゴリズムの入力言語として，以下のような特徴を持つ Pascal 風言語を想定している．この言語は現実の言語に比べ，簡単化されてはいるが，本質的な構造をすべて含んでおり，一般の言語に拡張できる．その言語には文として条件文 (if)，代入文，繰り返し文 (while)，入力文 (readln)，出力文 (writeln)，手続き呼びだし文，複合文 (begin–end) がある．変数の型としてはスカラ型のみでポインタ型はない．プログラムは, 大域変数宣言，手続き (および関数) 定義，メインプログラムからなり，ブロック構造はない．手続き内では，内部で宣言された局所変数と仮引数変数および大域変数のみが参照可能で，他の手続き内の局所変数は参照できない．手続きは, 自己再帰的および相互再帰的に定義可能であり，その引数は, 値渡しで扱われる．

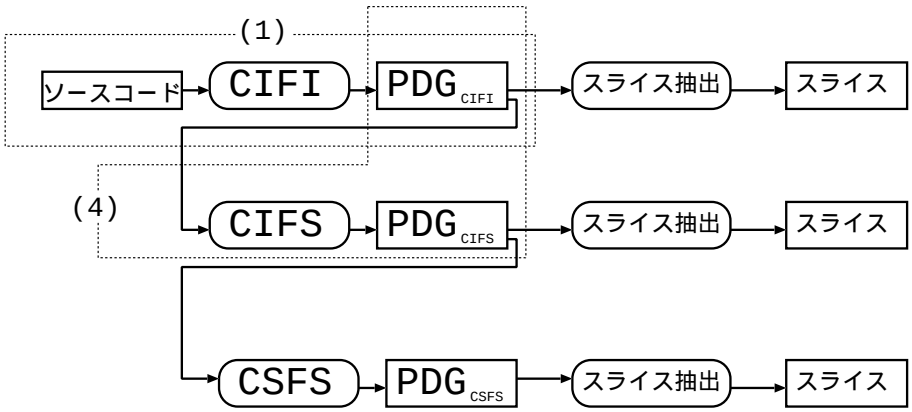
(1)
ソースコード
CIFI
PDG CIFI
スライス抽出
スライス
(4)
CIFS
PDG CIFS
スライス抽出
スライス
CSFS
PDG CSFS
スライス抽出
スライス

図 9: 提案する手法

3　CIFIグラフ構築アルゴリズム

本節では，『CIFIグラフ構築アルゴリズム』の出力となるPDGの構造およびCIFIグラフ構築アルゴリズムについて述べる．このアルゴリズムは関数内アルゴリズムと関数間アルゴリズムの2種類に分類できる．

3.1　プログラム表現

本節で述べるCIFIグラフ構築アルゴリズムの生成するPDGは，3.2 節，3.3 節で述べるCIFSグラフ構築アルゴリズムの入力となる構造を持つ．このPDGを従来のPDG[20]と区別するために，CIFI-PDGと呼ぶ．CIFI-PDGは一般的なPDGと同様に，プログラム中の文を表す節点と依存関係 (表2) を表す辺から構成される．また，関数間の依存関係を表現するために， 表3 に示す特殊節点を用意する． 図10 に示すソースコードに対するCIFI-PDGを 図11 に示す．

表 2: 依存辺

関数内の依存関係を表す辺	
def-def	変数の定義-定義関係
def-ref	変数の定義-参照関係
control dependence	制御依存関係
func-dependence	関数呼出し文と特殊節点間の関係
関数間の依存関係を表す辺	
parameter	実引数と仮引数の関係
func-return	関数の返値と関数呼出し文の関係

表 3: 特殊節点

global-def-in	手続き外からの大域変数の影響を内部へ伝えるための節点で，その手続き内で定義/参照している大域変数に対して，ひとつずつある
global-def-out	手続き内で定義された大域変数の影響をその外へ伝えるための節点で，定義/参照しているの大域変数に対して，ひとつずつある
global-out	関数/手続き呼出しによって関数が実行された後に，その関数から制御が戻って来るときに，その関数内の大域変数に与えた影響を呼出した関数(手続き)に与えるための節点である．
parameter-in	関数/手続き呼出しの引数を表現するための節点である．各引数ごとに用意される．
parameter-out	関数/手続き呼出しの引数が，呼ばれた関数により，影響を受けている場合(例えば，参照渡しで値が変更されたような場合)，呼んだ関数に制御が戻ってきたときに，その影響を伝えるための節点である．
actual-in	手続き/関数の実引数の手続き/関数内部への影響を伝えるための節点である．
actual-out	手続き/関数の実引数の手続き/関数外部への影響を与えるための節点である．
func-return	関数の戻り値を通して伝わる影響を検出するための節点で，各関数呼出しにひとつずつある．
func-exit	関数の戻り値を通して伝わる影響を検出するための節点で，各関数にひとつずつある．

```
...
a=4              f(x){
b=f(a)              f=x+2
print(b)         }
...
```

図 10: サンプルプログラム

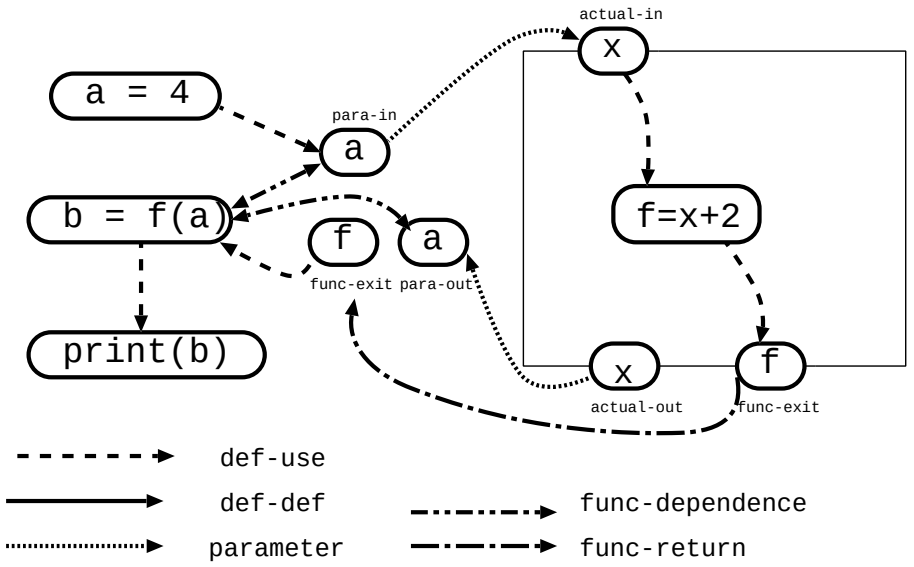

図 11: CIFI-PDG の例

3.2 関数内アルゴリズム

関数ごとの CIFI-PDG の構築方法について述べる．制御フローを考慮した解析では，文を複数回解析する可能性があるが，本手法のように制御フローを考慮しない解析では，各文を一回だけ解析するだけで良い．CIFI-PDG を構築する際のアルゴリズムは以下のようになる．

アルゴリズム AnalyzeFunction

Input: 関数 F

Output: F の CIFI-PDG

Step 1 . 以下の集合の初期設定を行なう．

$$DefVariable = \phi$$

$DefVariable$ は，ある文に到達する変数定義からなる集合である．

Step 2 . 関数内の全ての文に対して，(1) ~ (4) を実行する．全ての文を実行すれば，その関数の解析は終了する．さらに関数を解析終了後に，その関数内で定義した大域変数の global-def-out 節点を作る．

1. 変数を定義している文なら，まず $DefVariable$ を検索する．同じ変数の定義が存在する場合には，その節点に def-def 辺を引く．存在しない場合には，その関数に属する global-def-in 節点を作成し，def-def 辺を追加する．その後，その変数と節点番号の組を $DefVariable$ に追加する．
2. 参照変数が存在する文の場合には，$DefVariable$ からその変数に対応する組を検索する．もし存在する場合には，その節点に def-ref 辺を追加する．ない場合には，その関数に属する global-in 節点を作成し，def-ref 辺を追加する．
3. 解析対象の文が，if 文/while 文により入れ子構造を有する場合には，if 文/while 文から，その文に対して制御依存辺を引く．また，入れ子の深さが 2 以上の場合には，直接，入れ子になっている if 文/while 文にのみ制御依存辺を引く．
4. 解析対象の文が call 文であるとき，関数を越えて依存関係が存在する可能性があるために， 3.1 節で示した特殊節点及特殊辺が必要になる．パラメータが参照渡しである場合には，関数から出るときに値が変更されているかもしれない，また，大域変数等も関数内で変更されている可能性がある．大域変数に関しては，関数呼出し文を解析する時点に関数本体の解析が終了している場合には，関数本体から大域変数の情報を得て，関数呼出し文に付加する．もし解析が終了していない場合には，関数本体終了後に大域変数の情報を与え関数呼出しによる大域変数への影響を無視し，その関数呼び出し文以降に，大域変数を参照，定義する場合には，その関数呼び出し文で大域変数が変更されると仮定する．具体的には，global-in 節点及び global-out 節点を用意する．これらの節点は， 3.3 節のプログラム内アルゴリズムにおいて存在するべきかどうかが判定される．

3.3 関数間アルゴリズム

本節では関数ごとの CIFI-PDG を連結し，プログラムの CIFI-PDG を構築する方法について述べる．本手法では関数間の呼び出し関係を表す Call Graph [4]を用いて，関数ごとの CIFI-PDG を連結する．このとき，関数呼び出し文の特殊節点 (実引数，大域変数) と関数本体の特殊節点 (仮引数，大域変数) を辺でつなぐ．これにより関数間のデータ依存関係を表現できる．

関数内アルゴリズムにおいて関数呼び出し文以降の文の解析では，大域変数を定義/参照すると，関数呼び出し文の特殊節点 (大域変数) を用意した．しかし，この時点で実際に関数内で値が再定義されているか，値の変更はないかどうかが判定できるため，もし関数内で

[4]各節点は手続き名で，手続き間の『呼ぶ-呼ばれる』関係を，呼ぶ側から呼ばれる側への有向辺で表したもの．

値が変更されていない場合には，その大域変数に関する特殊節点は削除する．このとき，削除された節点に引かれている辺は，ソースコード中で前に存在する文に対応する節点へと引き直される．もし，解析対象のプログラムが再帰を含む場合には，再帰を構成する関数群の global-def-in 節点及び global-def-out 節点の数が収束するまで，繰り返し上記の作業を行う．

Call Graph を利用して，関数ごとの CIFI-PDG に連結するときのアルゴリズムを以下に示す．

アルゴリズム PdgMerge

Input: プログラム P 中の全ての関数 F の CIFI-PDG

Output: プログラム P の CIFI-PDG

Step 1 . プログラム P の Call Graph 上で強連結成分を抽出して，その集約グラフを作り，その根から深さ優先探索して，ポストオーダーで順序付けする (ただし，一回の探索ですべての節点に到達できないときは，未到達の節点がなくなるまで繰り返し，すべての節点を順序付けする)．

Step 2 . 各強連結成分内の要素を任意の要素から深さ優先探索して，ポストオーダーで順序付けする．

Step 3 . Step 1 で決定した順序で強連結成分をひとつずつ選び，関数単位の CIFI-PDG を連結する．global-def-out 節点がある場合には，対応する global-out 節点と paramter 辺でつなぐ．しかし，対応する global-out 節点が存在しない場合もある．このときは対応する呼び出し文に global-out 節点を生成し，連結する．さらに，対応する呼び出し文においてもその変数の global-def-out 節点を生成し，その global-def-out 節点と呼び出し文の global-out 節点を def-ref辺でつなぐ．

例えば， 図 12 の場合には，関数 B は global-def-out 節点を持たないが，関数 C において変数 a が定義されていることを，関数 A に伝える必要がある．このとき上記のアルゴリズムを用いると連結する順序は，C⇒B⇒A となる．関数 C の global-def-out 節点の存在により，関数 B にも global-out，global-def-out 節点が作成され，最終的に関数 A にある関数 B の呼び出し文の global-def-in，global-def-out 節点にも parameter 辺が引かれる．このようにして，変数 a に関するデータ依存は表現できる．global-def-in 節点がある場合には，対応する global-in 節点と paramter 辺でつなぐ，しかし，対応する global-in 節点が存在しない場合もある．例えば， 図 13 の場合には，対応する呼び出し文に global-in 節点を生成し，連結する．さらに，対応する呼び出し文においてもその変数の global-def-in 節点を生成し，その節点と呼び出し文の global-in 節点を def-ref辺でつなぐ．

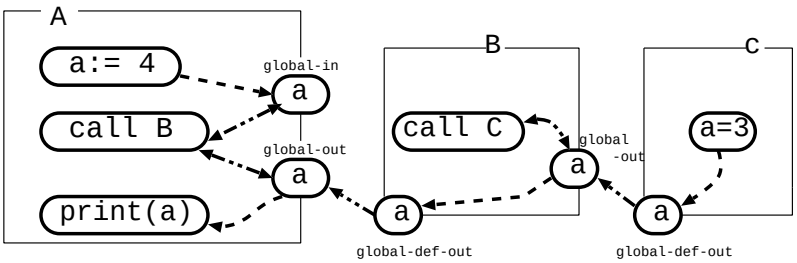

図 12: 関数間連結 (その 1)

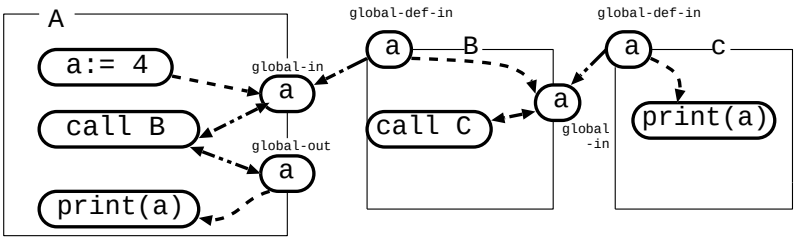

図 13: 関数間連結 (その 2)

4 CIFSグラフ構築アルゴリズム

本節では，3 節の CIFIグラフ構築アルゴリズムにより生成された CIFI-PDG を用いた CIFS 解析の方法について述べる．

4.1 方針

3 節で述べたように，プログラムを CIFI 解析によって CIFI-PDG に変換する際には，プログラム全体を対象として解析及び変換を行う．しかし，CIFSグラフ構築アルゴリズムでは，CIFI-PDG 全体を CIFS-PDG に変換するのではなく，必要な部分のみを変換する．必要な部分とは，あるスライシング基準に対するスライスに含まれる部分である．このようにする事で，抽出されるスライスに比例した時間でスライス計算が可能となる． 図 14 に部分的な変換について示す． 図 14 の左は，CIFI 解析により得られた CIFI-PDG を示している．次に，スライシング基準として文 $print(b)$ の変数 b を指定したときに，CIFI-PDG を用いてスライスを計算する． 図 14 の右に，抽出されたスライスに含まれる文を灰色にて示しているが，スライスに含まれる文に関する依存関係のみが再計算されている (太い実線は，CIFS 解析により計算されたデータ依存関係を表す)．以降の節では部分的な変換の実現方法，つまり，必要な部分 CIFS 解析方法について述べる．制御依存関係は CIFI 解析時に計算されているため，データ依存関係の計算が対象となる．

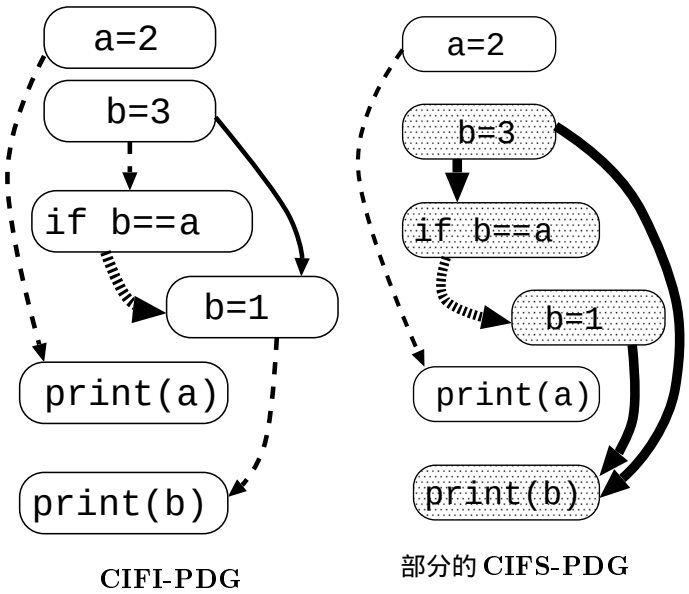

図 14: 部分的な CIFS 解析前後の PDG の例

4.2 データ依存関係の計算方法

ここではCIFI-PDGを用いて，文sの変数vとデータ依存関係のある文の集合Sを計算する方法について述べる．データ依存関係を計算するためには，変数vを定義している文を特定する作業が必要であるが，CIFI-PDGのdef-ref辺及びdef-def辺を利用することで特定することが可能である．基本的には，まず，文sに対応する節点nと節点nからdefref辺を逆方向に辿って到達できる節点n_1との間にデータ依存関係が存在するかを判定し，次に節点n_1からdefdef辺を逆方向に辿ることによって到達できる節点n_2と節点nとの間にデータ依存関係が存在するかを判定するというように，以降同様の作業を繰り返し行う．ただし，節点nで節点n'で定義する変数vを必ず参照するような場合には，節点n'以前に実行される節点の変数vの定義は節点nまで到達しないために，節点n'以前に実行される節点は節点nとデータ依存関係が存在しないことが自明であり，そのような節点n'に到達した時点で節点nのデータ依存関係の計算は完了する．また，節点nがループ文内にあるときには，上述のようにdefdef辺及びdefref辺を逆方向に辿るだけでなく，順方向に辿ることが必要となる．

1. 節点Vから辺を逆方向に辿る場合．
2. 節点Vから辺を順方向に辿る場合．

のそれぞれについて処理が必要となる．4.2.2節，4.2.3節，でそれぞれの場合のデータ依存関係の計算方法について述べる．

4.2.1 諸定義

ここでは4.2.2節，4.2.3節，で述べる処理を簡潔に述べるための定義を行う．

節点nから変数vに関するdefdef辺を順方向に辿ることで到達できる節点を$ForDD(n,v)$，逆方向に辿ることで到達できる節点を$BackDD(n,v)$と書く．defref辺に関しても同様に$ForDR(n,v)$，$BackDR(n,v)$と書く．$ForDD(n,v)$，$BackDD(n,v)$，$ForDR(n,v)$，$BackDR(n,v)$のそれぞれについて，もし対応する節点が存在しない場合にはNULLを値として持つ．

$BackDD(n,v)$からdefdef辺を逆方向に辿って節点n'に到達するまでの経路に含まれる節点のうち節点n'自身を除いた節点の集合を$AnalyzedDefV(v,n,n')$と定義する．

節点nからcontrol辺を逆方向に辿ることで到達できる節点，つまり，節点nと制御依存のある節点を$CD(n)$と書く．もし対応する節点が存在しない場合にはNULLとする．また，節点nからcontrol辺を逆方向に推移的に辿ることで到達できる節点の集合を$TransitiveCD(n)$と定義する．

関数$SameWhile(n,n')$を以下のように定義する．

Input: 節点 n 及び節点 n'

Output: TRUE 又は FALSE

Step 1 . $TransitiveCD(n)$ 及び $TransitiveCD(n')$ を計算する

Step 2 . $TransitiveCD(n) \cup TransitiveCD(n')$ が ϕ ならば FALSE , ϕ でないなら TRUE を返す .

ある節点 n に節点 n' の変数 v の定義が必ず到達する/到達しない可能性がある/必ず到達しないかを判定する関数 $ReachPossibility(n\ ,n')$ を以下のように定義する .

Input: 節点 n 及び節点 n'

Output:

MAY: 到達しない可能性があることを表す .
MUST: 必ず到達することを表す .
NOT: 必ず到達しないことを表す .

Step 1 . 実行可能性の判定 .

節点 n が実行されるとき節点 n' は以下の 3 種類のいずれかに当てはまる .

1. 必ず実行される .
2. 実行されない可能性がある .
3. 必ず実行されない .

(2) の実行されない可能性がある場合とは , 節点 n' が条件文内にあるような場合であり , (3) の必ず実行されない場合とは , 節点 n と節点 n' とが , 同じ分岐文の異なる節にある場合である . (3) の場合は NOT を出力し , 終了する . (1)(2) の場合には Step 2 を実行する .

Step 2 . 到達可能性の判定 .

Step 1 で実行される可能性が存在すると判定されたとしても , もし , 以下のような場合には節点 3 により節点 2 の変数 a の値は , 節点 5 には到達しない . このように , これまでに解析を行った節点により , 節点 n の定義が消去されるかどうかを判定する必要がある .

```
1       if(b>0){
2           a=3;
3           a=4;
4       }
5       print(a);
```

具体的には，節点n'が実行されるとき，必ず実行されるような節点が$AnalyzedDefV(v, n, n')$に含まれるとき節点n'の変数vの定義は節点nまで到達しない．含まれるときには，NOTを出力する．それ以外には，Step 1で得られた値を出力する．

4.2.2 逆方向のデータ依存関係の計算方法

ここでは，文sに対応する節点nと節点nからdefref辺及びdefdef辺を逆方向に辿って到達できる節点を対象としたデータ依存関係の計算アルゴリズムを示す．

アルゴリズム

GET BACKWARD DATADEPEPNDENCE

Input:

節点と変数の組，$< n, v >$

Output:

文の集合$BackDDStatement$

Step 1．初期化．

$BackDDStatement \leftarrow \phi$
$n' = BackRD(n, v)$

Step 2．関数を越えるデータ依存関係の処理．

もし節点nがglobal-def-in節点であれば，parmeter辺を辿る．到達する節点をn'とし，Step 6へ．

Step 3．関数$ReachPossibility(n, n')$の実行．

もし返り値が

1. MAY $\rightarrow$ Step 4．
2. MUST $\rightarrow$ Step 6．

3. NOT → Step 5 .

Step 4 . 節点n'と節点n間にはデータ依存関係が存在するので，集合$ForDDStatement$に節点n'を追加する .

$BackDDStatement \leftarrow n'$

Step 5 . $n' = BackRD(n', v)$

Step 3へ .

Step 6 . 逆方向へのデータ依存関係の計算終了 .

$BackDDStatement \leftarrow n'$

$BackDDStatement$が節点nとの間にデータ依存関係の存在する節点の集合となる .

4.2.3 順方向のデータ依存関係の計算方法

ここでは，文sに対応する節点nと節点nから defref辺及び defdef辺を順方向に辿って到達できる節点を対象としたデータ依存関係の計算アルゴリズムを示す . 順方向に辿る必要があるのは，節点nがループ文内に存在するとき，つまり，集合$TransitiveCD(n)$にループ文が含まれるときである .

アルゴリズム

GET FORWARD DATADEPEPNDENCE

Input:

節点と変数の組，$< n, v >$

Output:

文の集合$ForDDStatement$

Step 1 . 初期化 .

$ForDDStatement \leftarrow \phi$

$n' = ForDD(BackRD(n, v), v)$

n''=n'

Step 2 . $SameWhile(n, n')$の実行 .

返り値がTRUEならば，

$n'' = n'$

$n' = ForDD(n', v)$

とし，Step 2へ．FALSEならば，Step 3へ．

Step 3．関数$ReachPossibility(n, n'')$の実行．

もし返り値が

1. MAY → Step 4．
2. MUST → Step 6．
3. NOT → Step 4．

Step 4．節点n''と節点n間にはデータ依存関係が存在するので，集合$ForDDStatement$に節点n''を追加する．

$ForDDStatement \leftarrow n''$

Step 5．$n'' = BackRD(n'', v)$

$SameWhile(n, n')$を実行し，返り値が

- TRUE → Step 3．
- FALSE → Step 6．

Step 6．順方向へのデータ依存関係の計算終了．

$ForDDStatement \leftarrow n''$

$ForDDStatement$が節点nとの間にデータ依存関係が存在する節点の集合となる．

4.3 アルゴリズム

スライス計算時には，スライシング基準からデータ依存関係を再計算し，スライシング基準とデータ依存関係のある文との間にデータ依存関係のある文を再計算するというように，再帰的にデータ依存関係を計算することで，スライスが計算できる．まとめると以下のようなアルゴリズムになる．

アルゴリズム

GET SLICE

Input:

プログラムPのCIFI-PDG

スライシング基準(s, v)

Output:

スライス
部分的に CIFS 解析された CIFS-PDG

Step 1 . 以下の集合を初期化する ．

$$Slice \leftarrow \{(s, v)\}$$
$$Sliced \leftarrow \phi$$

$Slice$ は文と変数の組からなる集合であり，解決すべきデータ依存関係を表す．$Sliced$ は文からなる集合であり，スライスに含まれる文を表す ．

Step 2 . $Slice$ 内の各要素について以下を繰り返し実行する . このとき注目する要素を (S, V) とする ．$Slice = \phi$ ならば終了する ．

1. 文 S の変数 V に関してデータ依存関係のある文を計算するために ,Get Forward DataDepepndence 及び Get Back DataDepepndence を実行する ．Get Forward DataDepepndence の出力 $ForDDStatement$ であり，Get Back DataDepepndence の出力 $BackDDStatement$ である ．このとき

   $S' \in ForDDStatement \cup BackDDStatement$

   であるような全ての文 S' について，文 S' で使用している変数を V' とし，全ての (S', V') に対して，

   $Slice \leftarrow (S', V')$

   とする ．

2. 文 S と制御依存関係のある文を S' とすると，

   $Slice \leftarrow (S', V')$

   とする ．

3. $Slice \leftarrow Slice$ - (S, V)
   $Sliced \leftarrow S$

Step 3 . スライスは集合 $Sliced$ に含まれる文となる ．

5　実行効率

5.1　計算量

この節では 4 節， 3 節で紹介したアルゴリズムの複雑さについて述べる．本アルゴリズムは，AnalyzeFunction と PdgMerge から成る CIFI-PDG 構築部と，Get Slice から成るスライス計算部に分けられるので，複雑さについても両者を分けて考える．

5.1.1　CIFI-PDG を作るコスト

ソースコードから CIFI-PDG を作るための解析に関わる要素を以下の表の様に定義する．

PDG 作成に関わる要素	
P	手続きの総数
G	大域変数の総数
L	手続きの局所変数の数の最大値
S_i	手続き呼び出しの総数
S_t	文の総数

CIFI-PDG を構築するためには AnalyzeFunction と PdgMerge のそれぞれの実行に要する時間の和となる．以下，それぞれの複雑さを考える．

アルゴリズム AnalyzeFunction

プログラム中のすべての文を 1 回だけ解析する．プログラム中のすべての文を一回解析するのに要する時間は, 文の数に比例する．ひとつの文の解析には定数回の集合演算が必要である．一回の集合演算にかかる時間が集合に含まれる要素の数に比例すると仮定すると，ひとつの文の解析の時間は大域変数と局所変数の数の和に比例する．よって，$O(S_t \cdot (G + L))$ となる．

アルゴリズム PdgMerge

Step 1 はプログラムの CG の節点の数に比例する．また，Step 2 は有向辺に比例する．節点の数は P で, 有向辺の数は S_i で, それぞれ抑えられる. Step3 は各手続きにおいて大域変数の数のだけ行われる．よって，$O(P + S_i + P \cdot G)$ となる．

以上から，CIFI-PDG 構築アルゴリズムの時間計算量は $O(P + S_i + P \cdot G + S_t \cdot (G + L))$ である

5.1.2 CIFI-PDG を用いてスライスを計算するコスト

PDG に関わる要素を以下のように定義する.

PDG 上でのスライスの計算に関わる要素	
G	大域変数の総数
L	手続きの局所変数の数の最大値
S_t	文の総数

CIFI-PDG に対して CIFS 解析を行いスライスを計算するとき，全ての節点がスライスに含まれる場合に最も多くの計算量が必要になる．この場合には，全ての節点において，その節点の全ての参照変数のデータ依存関係を解決する必要がある．また，ある参照変数のデータ依存関係を解決するときに，最悪時には全ての節点との間でデータ依存関係が存在するかを判定する必要がある．以上から，CIFI-PDG を用いたスライス計算に要するコストは $O(S_t \cdot (G + L) \cdot S_t)$ である．

5.2 実装

本アルゴリズムを文献 [20] のアルゴリズムを実装した文献 [17] のシステムに組み込んだ．ソースコードは C で記述し，ユーザインタフェースには Tcl/Tk を用いている．アルゴリズム AnalyzeFunction は約 1500 行，アルゴリズム PdgMerge は約 500 行，アルゴリズム Get DataDepepndence And Slice は約 2000 行であり，システム全体では約 17500 行である．

このシステムを用いて 表 4 に示すサンプルプログラムに対してスライスを抽出し，従来手法と本手法を比較する．以下の点について比較を行う．

- 従来手法の PDG 構築時間と本手法の CIFI-PDG 構築時間．(図 15 の (1) と (3))
- 従来手法により構築された PDG からスライスを抽出するのに要する時間と本手法により構築された CIFI-PDG からスライス抽出にするのに要する時間．(図 15 の (2) と (4))
- あるソースコードがあり，そのソースコードの依存グラフが存在しないときに，ある文のある変数に関するスライスを抽出するのに要する時間．(図 15 の (1)+(3) と (2)+(4))

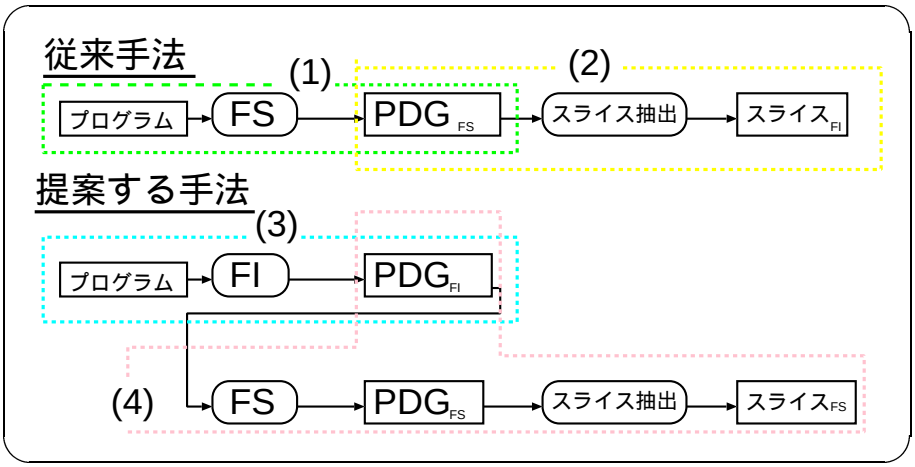

図 15: 比較部分

表 4: テストプログラム

	行	内容
P_1	38	バブルソート
P_2	247	予約問題
P_3	305	酒屋問題
P_4	610	拡張酒屋問題

表 5: 依存グラフ構築に要した時間 (ms)

プログラム	P_1	P_2	P_3	P_4
プログラムサイズ	38 文	247 文	305 文	610 文
従来の手法 (1)	12	474	1116	6650
本手法 (3)	3	23	36	83

表 6: 依存グラフに対するスライス抽出に要した時間 (ms)

プログラム	P_1	P_2	P_3	P_4
プログラムサイズ	38 文	247 文	305 文	610 文
スライスサイズ	23 文	104 文	151 文	141 文
従来の手法 (2)	0.1	0.8	1	2
本手法 (4)	0.3	2	4	4

表 7: ソースプログラムに対するスライス抽出に要した時間 (ms)

プログラム	P_1	P_2	P_3	P_4
プログラムサイズ	38 文	247 文	305 文	610 文
スライスサイズ	23 文	104 文	151 文	141 文
従来の手法 (1+2)	12	474	1117	6652
本手法 (3+4)	3	25	40	87

表 5 に，従来手法の PDG 構築時間と本手法の CIFI-PDG 構築時間， 表 6 に，従来手法により構築された PDG からスライスを抽出するのに要する時間と本手法により構築された CIFI-PDG からスライス抽出にするのに要する時間， 表 7 に，あるソースコードがあり，そのソースコードの依存グラフが存在しないときに，ある文のある変数に関するスライスを抽出するのに要する時間をそれぞれ示す (Celeron 300A MHz, 128M 上での実行時間)．表 6 ，表 7 のスライスサイズは，典型的なスライシング基準[5]に対してスライスを計算したときに，スライスに含まれる文の数を表す．

5.3 考察

依存グラフ構築時間 (表 5) は，従来手法では CIFS 解析を行っているため，各文を複数回解析する必要があるが，本手法では CIFI 解析を行っているために，各文を 1 回だけ解析するだけでよいので従来手法と比べると非常に短くなっている．また，依存グラフ構築時間は，本手法ではプログラムサイズに依存するが，従来手法では，プログラムの複雑度 (ループ文，再帰呼び出し) に依存する．

[5] 一般的に，フォールト位置特定の際には，出力文の誤った値である変数をスライシング基準に指定する．

依存グラフからのスライス抽出時間 (表 6) は，従来手法ではグラフ構築時に制御フローを考慮した解析を行っているので，依存辺を単純に辿るだけでよいが，本手法では依存グラフ構築時には，制御フローを考慮していないために，スライス抽出時に制御フローを考慮した解析を行いながらスライスを抽出するので，従来手法よりも時間を要する．

ソースコードから依存グラフを構築し，スライス抽出するのに要する時間 (表 7) は，従来手法では，スライスに含まれない文まで制御フローを考慮して解析するためが，本手法ではフローを考慮しない解析後に，スライスに含まれる可能性のある文に関してのみフローを考慮して再解析を行う．制御フロー解析の対象とソースコード全体でなく，スライスに含まれる可能性のある部分に限定する本手法は有効であると確認できた．

6 まとめ

本研究では，簡略化の度合の異なる依存解析アルゴリズムの生成する依存関係情報を共有，再利用し，プログラムスライス抽出の効率を制御するための方法を提案した．また本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．

従来のPDGを利用したスライス計算アルゴリズムでは，最初にソース全体に対して正確な依存解析を行う必要があったが，本手法では，最初は不正確な(時間をあまり要さない)依存解析を行い，必要な部分のみを正確な依存解析を行うことが可能である．

今後の課題としては，本研究の対象言語はポインタ型変数をもたないものであるが，一般的な言語への本手法の適用可能性を考えること，また，スライス計算対象のプログラムにより，どの依存解析を行う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こと等があげられる．

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 基礎工学部 情報科学科 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 講師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 基礎工学部 情報科学科 井上研究室の皆様に深く感謝致します．

参考文献

[1] Atkison, D. C. and Griswold, W. G.: "The Design of Whole-Program Analysis Tools", In *Proceedings of the 18th International Conference on Software Engineering*, pp. 16–27(1996).

[2] Fiutem, R. , Tonella, P. , Antoniol, G. and Merlo, E.: "Variable Precision Reaching Definitions Analysis for Software Maintenance", In *Proc. of the Euromicro Working Conf. on Soft. Maintenance and Reengineering*, pp. 60–67(1997).

[3] Griswold, W. G. and Notkin, D.: "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool", *IEEE Transactions on Software Engineering*, 21(4):275–287(1995).

[4] Harrold, M. J. and Ci, N.: "Reuse-Driven Interprocedural Slicing", In *Proceedings of the 20th International Conference on Software Engineering*, pp. 74–83(1998).

[5] 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).

[6] 地平, 西松, 楠本, 井上: "関数呼出し情報を用いたスライスサイズの削減のための一手法", 電子情報通信学会論文誌 D-I(条件付採録).

[7] Krinke, J. and Snelting, G.: "Validation of Measurement Software as an application of Slicing and Constraint Solving. ", *Computer Science Report 98-01*(1998).

[8] Livadas, P. E. and Croll, S.: "Program slicing", *Technical Report SERC-TR-61-F, Software Engineering Research Center*(1992).

[9] Michael, D. E.: "Practical fine-grained static slicing of optimized code ", *Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA*(1994).

[10] Nishimatsu, A. , Jihira, M. , Kusumoto, S. and Inoue, K.: "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", In *Proceedings of the 21th International Conference on Software Engineering*(1999)(to appear).

[11] 西松, 楠本, 井上: "保守プロセスに対するプログラムスライスの実験的評価", 信学技報, Vol. 97, No. 630, pp. 79–86(1998).

[12] 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”, 信学技報, Vol. 98, No. 85, pp. 17–24(1998).

[13] 西松, 楠本, 井上: “効率と精度を考慮したプログラムスライス抽出法の提案”, 信学技報, Vol. 98, No. 559, pp. 9–16(1999).

[14] 西松, 楠本, 井上: “保守プロセスに対するプログラムスライスの実験的評価”, 電子情報通信学会論文誌 D-I(採録決定).

[15] 西松, 西江, 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”, 電子情報通信学会論文誌 D-I(投稿中).

[16] Ottenstein, K. J. and Ottenstein, L. M.: “The Program Dependence Graph in a Software Development Environment”, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5), pp. 177–184(1984).

[17] 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).

[18] 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).

[19] 高田，佐藤 ，井上: “プログラム依存グラフの効率的な更新手法”, 電子情報通信学会論文誌, Vol. J81-D-I, No. 3, pp.253-260(1998).

[20] 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).

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

[22] Weiser, M.:“Programmers use slices when debugging”, *Communications of the ACM*, Vol. 25, No.7, pp. 446-452(1982).

[23] Weiser, M.: “Program Slicing”, *IEEE Trans. on Soft. Eng.*, Vol.10, No. 4, pp. 352-357(1984).

[24] Weise, D. , Crew, R. F. , Ernst, M. D. , and Steensgaard, B.: “Value Dependence Graphs: Representation Without Taxation”, *ACM Symposium on Principles of Programming Languages*, pages 297-310(1994).

[25] "The Wisconsin Program-Slicing Tool 1.0, Reference Manual", Computer Sciences Department, University of Wisconsin-Madison, (1997).