効率と精度を考慮したプログラムスライスの抽出法の提案

西松 顯　井上 克郎

大阪大学大学院基礎工学研究科情報数理系専攻
〒560-8531 大阪府豊中市待兼山町 1-3 基礎工学研究科 井上研究室
Phone: 06-850-6571　Fax: 06-850-6574
E-mail: {a-nisimt, inoue}@ics.es.osaka-u.ac.jp

あらまし

本研究では，複数の効率の異なるプログラムスライス抽出技法の生成する依存関係情報を共有，再利用し，効率と精度を考慮したプログラムスライス抽出技法を提案する．従来の手法では複数の依存関係解析アルゴリズムを持つような場合，アルゴリズム間で依存関係情報を共有，再利用することができなかった．ある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．本手法では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，複数の依存関係解析アルゴリズムにおいてスライス計算が効率良く行うことが可能である．本手法は効率及び精度の異なる 3 種類の依存関係解析アルゴリズムを対象としている．

キーワード　プログラムスライス，効率，精度

Proposal of a Slicing Algorithm and Its Efficiency and Accuracy Trade-off

Akira Nishimatsu and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka-shi, Osaka 560-8531, Japan
Phone: +81-6-850-6571　Fax: +81-6-850-6574
E-mail: {a-nisimt, inoue}@ics.es.osaka-u.ac.jp

Abstract

In this paper, we propose a new slicing algorithm considering efficiency and accuracy trade-off. This algorithm consists of three kinds of the dependency analysis algorithm with the statements in the source program. The dependency information is shared and reused between these dependency analysis algorithms. By using our technique, a program slice is efficiently computed with the differrent dependency analysis.

Key words　program slice, efficiency, accuracy

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)．スライスの技法は Mark Weiser [11] 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 [3, 9].

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，非構造化な文 (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ような事も生じて来る．そこで，これまでに依存関係の正確性を向上させるような研究が多数されている (Calling Context[3], 関数間制御依存関係 [7]，statement side-effect[2, 5] 等)．

依存関係の正確性を向上させるにつれ，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で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もされている [12]．そこで，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事が必要になってきた [1]．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．さらに，Tonella らは，3 種類の正確性の異なるアルゴリズムを有するシステムを開発し、ユーザにスライス計算アルゴリズムを指定できるようにした [4]．しかし，これらの研究においては，各依存解析方法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方法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，複数の効率の異なるプログラムスライス抽出アルゴリズムの生成する依存関係情報を，各アルゴリズム間で共有，再利用し，効率と精度を考慮したプログラムスライス抽出法を提案する．各スライス抽出アルゴリズム間で共有な可能なデータ構造を定義し，その構築方法を述べる．また，あるスライスアルゴリズムによって生成された依存関係情報を別のスライスアルゴリズムで利用する方法についても述べる．

以降，2．ではスライスの計算手順と計算効率について述べる．3．では共有可能な依存関係情報のデータ構造および構築法について述べる．4．では共有可能な依存関係情報を利用したスライス計算方法について述べる．最後に，5．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

2.1 スライス計算手順

図1に通常多く用いられるスライスの計算手順を示す．スライス計算手順は，2 つのフェーズに分けられる．以降，2 つのフェーズについて述べる．

2.1.1 Phase 1

フェーズ 1 では，ソースコードを入力とし，依存関係解析を行い，プログラム依存グラフを出力する．プログラム依存グラフ (Program Dependence Graph．以降，PDG と呼ぶ) とは，プログラム中の文間の依存関係をグラフに表現したもので，各節点は文を，各辺は依存関係を表す．依存関係には制御依存関係とデータ依存関係の以下の 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．

- 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．

2.1.2 Phase 2

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，まず,『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．指定された文に対応する PDG 上の節点から，推移的に依存辺を辿って到達できる節点に対応する文の集合がスライスとなる．

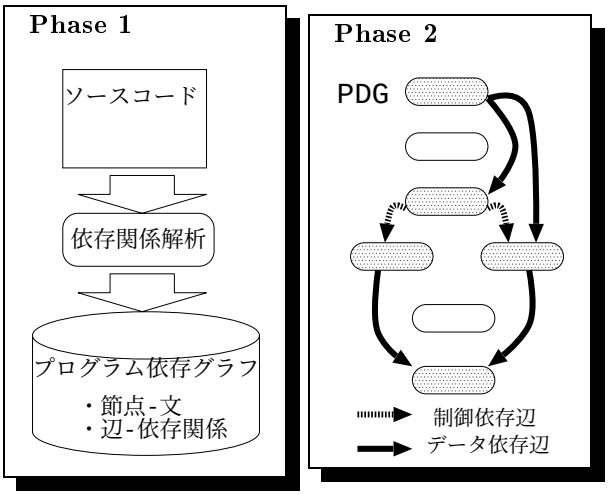

図 1: スライス計算手順

2.2 効率

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 [6]．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事を言い，スライスの安全性を保ちつつ正確性を犠牲にしてい向上をはかるものとして，(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．

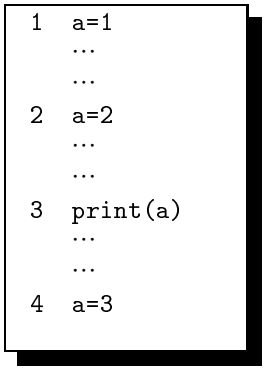

図 2: 制御フロー

```
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 3: Calling-Context

図2及び図3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2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3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[1]の依存関係解析アルゴリズムが存在する [4]．

(1) CSFS(Context Sensitive Flow Sensitive)

(2) CIFS(Context Insensitive Flow Sensitive)

(3) CIFI(Context Insensitive Flow Insensitive)

[1] 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 [4].

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

3 提案する手法

3.1 従来の手法の問題点

文献 [4] では，2.2節の 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 4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．

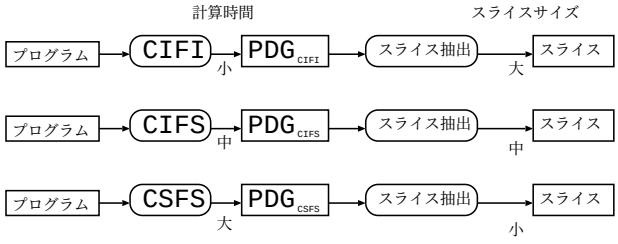

図 4: 従来手法

3.2 研究の目的

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．[2].

(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 $\Rightarrow PDG_{CSFS}$

[2] 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．

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 4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 [3] 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 5)．以降，4節で図 5の (1) について述べ，5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．

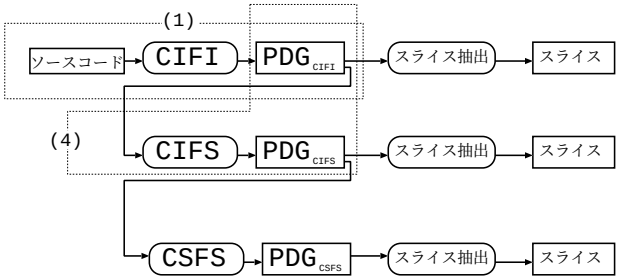

図 5: 提案する手法

4 CIFIグラフ構築アルゴリズム

ここでは,『CIFIグラフ構築アルゴリズム』の出力となる PDG の構造について述べ，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．

4.1 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，次節で説明する 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 を一般的な PDG と区別するために，CIFI-PDG と呼ぶ．CIFI-PDG は 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 6に示すソースコードに対する CIFI-PDG を図 7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 6: サンプルプログラム

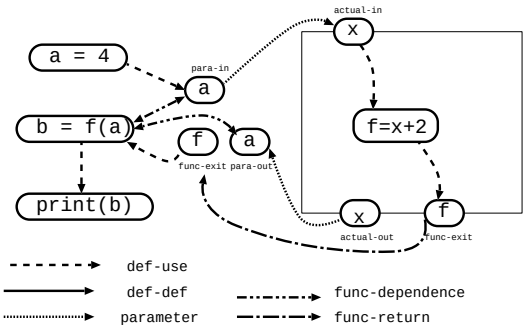

図 7: CIFI-PDG の例

4.2 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．

$$DefVariable = \phi$$

集合 $DefVariable$ 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-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 $DefVariable$ 集合に追加する．

(2) 参照変数が存在する文の場合には，$DefVariable$ 集合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，4.1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，関数呼び出し文に遭遇した時点では無視する．その関数呼び出し文以降に，その関数呼び出し文により大域変数を参照，定義する場合には，その関数呼び出し文に global-in 節点及び global-out 節点を用意する．これ

らの節点は，4.3節のプログラム内アルゴリズムにおいて処理される．

4.3 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 [3]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，ソースリスト中で前に存在する文に対応する節点へと引き直される．本アルゴリズムは,『制御フローを考慮しない』となっているが，関数間のフローの制御に関してはCall Graphで判定できる範囲で制御フローを考慮している．しかし,『全ての制御フローを考慮する』場合と比較すると，再帰等を考慮しなくて良いため，関数間にまたがる依存関係の解析は1回だけで良い．

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．

[3]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．

例えば，図8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9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．

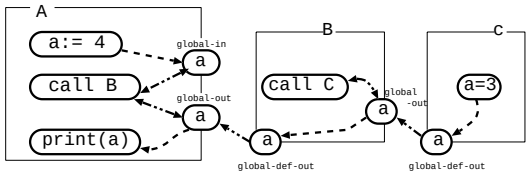

図 8: 関数間連結(その1)

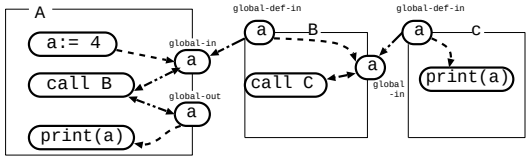

図 9: 関数間連結(その2)

5 CIFSグラフ構築アルゴリズム

本節では，4節により生成されたCIFI-PDGを用いたCIFS解析の方法について述べる．

5.1 方針

4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，本アルゴリズムでは，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10の部分的な変換について示す．図10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10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).

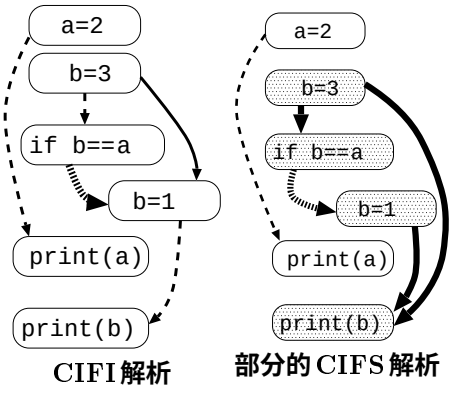

図 10: スライス計算手順

5.2 アルゴリズム

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，ここではデータ依存関係の計算が対象となる．CIFI-PDG を用いて，文 s の変数 v とデータ依存関係のある文 s' を計算する場合を考える．まず，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 辺を逆方向に辿ることで特定できる．ただし，図 10の様に文 $print(b)$ から def-ref 辺を辿ることによって到達可能な文 $b = 1$ は，if 文内にあるために，それ以前の変数 b の定義を消去することができない．そのために文 $b = 1$ 以前で変数 b を定義している文を特定する必要がある．これは CIFI-PDG 上の def-def 辺を辿れば特定できる．図 10では，文 $b = 3$ に到達できる．文 $b = 3$ は，それ以前の変数 b の定義を消去するので，文 $print(b)$ は，文 $b = 3$，文 $b = 1$ で定義されている変数 b を参照して，つまり，この 2 文との間にデータ依存関係があることがわかる．

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．これまでをまとめると以下のようになる．

アルゴリズム

GET DATADEPEPNDENCE AND 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．

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

集合 $Slice$ は文と変数の組からなる集合，集合 $Sliced$ は文からなる集合である．

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 CIFI-PDG を用いて計算する．このときデータ依存関係のある文を S'，また，文 S' で使用している変数を V' とすると，全ての (S', V') を集合 $Slice$ に追加する．ただし，集合 $Sliced$ に文 S' が存在する場合には追加しない．

(2) 文 S と制御依存関係のある文に関しても上記と同様に集合 $Slice$ に追加する．ただし，集合 $Sliced$ に文 S' が存在する場合には追加しない．

(3) 要素 (S, V) を集合 $Slice$ から除き，集合 $Sliced$ に文 S を追加する．

Step 3. スライスは集合 $Sliced$ に含まれる文となる．

6 考察

本節では，本手法の効率および精度について述べる．図 4に示す従来手法によるスライス計算と，図 5に示す本手法によるスライス計算とを比較する．

- CIFI 解析
  従来の CIFI 依存解析アルゴリズムと 4節で述べた ANALYZEFUNCTION，PDGMERGE アルゴリズムとを比較する．本手法では，生成された PDG が別の依存関係解析アルゴリズムにより再利用されるために，PDG の構造は従来のものと異なるが，各手法により生成される PDG が含む依存情報は同じものであるため，PDG 構築に要する時間および抽出されるスライスの精度は同じである．

- CIFS 解析
  従来の CIFS 依存解析アルゴリズムと本手法の GET DATADEPEPNDENCE AND SLICE アルゴリズムとを比較する．従来手法では，ソースコードを入力とするためにプログラム全体の依存解析を CIFS を用いて行う．しかし，本手法では，CIFI で生成された PDG を入力とするため，CIFS を行う部分はプログラムの一部となる．PDG の構築を行った後にスライス抽出を行うという手順を考えると，必要な部分に比例した時間を要する本手法の方が，従来手法よりもスライスシステムの構築には適していると思われる．ただし従来手法では PDG を一度構築した後は，スライス抽出時に，あまり時間を要さない．しかし，本手法においては必要な部分のみを正確に解析し，解析時に得られ

た正確な依存関係情報はPDGに付加されるため，スライスを計算する度に，新たにCIFS解析を行う．しかし，解析が必要な部分はスライスをとるごとに減少し，あまり時間を要さなくなる．現在，本手法をシステムに実装している段階であるため，効率に関しては比較できないが，実際にスライスシステムでの利用を考えると，ユーザに使用しやすいものになると考えている．

7 まとめと今後の課題

本研究では，複数の効率の異なるプログラムスライス抽出アルゴリズムの生成する依存関係情報を共有，再利用し，効率と精度を考慮したプログラムスライス抽出法を提案した．本手法を用いることにより，ユーザが効率の異なるスライス計算アルゴリズムを利用する場合に，依存関係の再計算の必要がなくなる．さらに，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．

現在，本手法をこれまでに開発したシステム[8]に実装している段階であり，実装後はこれまでの手法[10]との比較を行う予定である．

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*, pages 16–27, 1996.

[2] Michael, D. E. “Practical fine-grained static slicing of optimized code ”. *Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA*, July 1994.

[3] 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).

[4] 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*, pages 60–67, Berlin, Germany, March 1997.

[5] Livadas, P. E. and Croll, S. “Program slicing”. *Technical Report SERC-TR-61-F, Software Engineering Research Center*, October 1992.

[6] 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).

[7] Harrold, M. J. and Ci, N. “Reuse-Driven Interprocedural Slicing”. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, Kyoto, April 1998.

[8] 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).

[9] 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).

[10] 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).

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

[12] “The Wisconsin Program-Slicing Tool 1.0, Reference Manual”, Computer Sciences Department, University of Wisconsin-Madison, August,1997.