論　文

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

西松　顯[†]　地平　稔[††]　楠本　真二[†]　井上　克郎[†††]

A Slice Reduction Method Using Function Activation Information

Akira NISHIMATSU[†], Minoru JIHIRA[††], Shinji KUSUMOTO[†], and Katsuro INOUE[†††]

あらまし
大規模なソフトウェアのデバッグや保守の際に，プログラム全体を参照するのではなく，プログラムの一部に参照範囲を限定することは有効である．参照範囲を限定する方法の一つとして，プログラムスライス技法(Program Slicing)が提案されている．静的スライスは，注目する変数の値に影響を与える可能性のある文の集合を抽出する．しかし，多くの場合には，実際には関係のない部分までスライスに含まれるため，十分に参照範囲を限定することができない．一方，動的スライスは，注目する変数の値に影響を与える可能性のある実行された文の集合を抽出する技法で，プログラムを実際に実行し，その際に得られる動的情報を利用するため，十分に参照範囲を限定することができる．しかし，プログラム実行時の動的情報取得のために多くの時間的・空間的コストを要する．本論文では，プログラムを実行して得られるわずかな動的情報と静的な解析とを組み合わせたプログラムスライス抽出技法を提案する．この技法により抽出されたスライスを，コールマークスライス(Call-Mark Slice)と呼ぶ．コールマークスライスの計算時には，まずプログラムのデータ依存関係と制御依存関係はあらかじめ静的に解析し，次にプログラム実行時に手続きと関数の呼び出しの記録を保存する．そして，これらの情報から変数の動的な依存関係を明らかにすることで，従来よりも効果的にスライスを抽出することができる．更に，我々はコールマークスライスを既に開発しているスライスシステムに実装し，その有効性の評価も行った．

キーワード　静的スライス，動的スライス，依存解析，実行時オーバーヘッド(Execution Overhead)

1. まえがき

大規模ソフトウェアのデバッグや保守はソフトウェア工学における研究の主要なテーマの一つである．大規模ソフトウェアのデバッグや保守を効率良く行う方法の一つとして，プログラム全体を参照するのではなく，知りたい情報に直接的または間接的に関係する部分を抽出することで参照範囲を限定する方法がある．この方法の一つとして，プログラムスライス(Program Slice．以降，単にスライスと略す)を利用した手法が提案されている[8], [19]．

これまでに我々は，デバッグや保守におけるスライスの有効性について実験的に評価を行っている[14]．この実験では，被験者を二つのグループに分け，一方のグループに属する被験者は従来のデバッグツールのみを用い，他方は従来のデバッグツールとスライスツールを用いてフォールト位置の特定を行った．このとき，それぞれのグループがフォールト位置を特定するのに要した時間を計測した．要した時間について統計的に分析すると，二つのグループ間の作業時間の間には有意な差があった．スライスを用いることで，プログラマが参照する範囲は限定され，限定された範囲にのみ集中すればよくなり，結果として，効率良くフォールト位置特定が行えると考えられる．

これまでに，スライスに関する研究は数多く行われている[7], [8]．注目する変数の値に影響を与える可能性のある文の集合を抽出したものを静的スライスと呼ぶ．一般的に静的スライスは，元のプログラムからプログラムの一部分を抽出する技法であるが，ソースコードを解析する際に，可能性のあるすべての入力と制御フローを考慮しているために，多くの場合において抽出されたスライスのサイズは大きくなる．プログラムにあるデータを与えた実行に対し，注目する変数

[†] 大阪大学 大学院基礎工学研究科，豊中市
Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560-8531 Japan
[††] 奈良先端科学技術大学院大学情報科学研究科，生駒市
Nara Institute of Science and Technology Department of Information Science 8916-5, Takayama-cho Ikoma-City Nara 630-0101, Japan

の値に影響を与える可能性のある実行された文の集合を抽出したものを動的スライスと呼ぶ[1], [9]．動的スライスでは，特定の入力に対する実行に基づいているため，実行されなかった文及び実行されなかった文にのみ依存する文は自動的に除かれ，一般的に静的スライスよりも小さくなる．しかし，動的スライスの計算には，動的な依存関係の解析に多くの記憶領域と実行のオーバーヘッドを要する．

本論文では，これらの問題点を解消するために新しいスライス技法(コールマークスライス(Call-Mark Slice))を提案する．この技法の特徴は，

（1）静的な解析による情報と関数呼出しのわずかな動的情報を組み合わせ，実行のオーバーヘッドを最小限におさえる．

（2）スライスとして抽出される分量は，静的スライスと動的スライスとの間に位置する．

我々は，コールマークスライスを，既に開発しているスライスシステム[15]に実装した．更に，コールマークスライスの有効性を評価するために，さまざまなサンプルプログラムをシステム上で実行し，データを計測した．その結果，コールマークスライスは静的スライスよりもかなり参照する範囲を限定できること，また，実行時に動的情報を集めるために必要な負荷は動的スライスよりもはるかに小さいことが分かった．

2．では静的スライスと動的スライスの概略を述べる．3．ではコールマークスライスについて述べる．4．では我々のシステムへのコールマークスライスの実装とサンプルプログラムの実行例について述べる．5．では他のプログラムスライス抽出技法との比較について述べ，6．でまとめと今後の課題について述べる．

2. 静的スライスと動的スライス

2.1 静的スライス

今，ソースプログラムp中の文s_1，s_2について考える．次の条件をすべ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．

次に以下の3つの条件をすべて満たすとき，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．

```
 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  Qube: 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1 ソースプログラム
Fig. 1 Pascal Source Program

- 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が少なくとも一つは存在し，さらに，そのパスにおいて文s_1から文s_2間に変数vを定義している文が存在しない．

この関係を$s_1 \xrightarrow{v} s_2$と表す．

プログラム依存グラフ(Program Dependence Graph，PDG)の節点は，プログラムに含まれる代入文，入出力文，条件判定文，手続き呼出し文など各文を表し，辺は文の間の上記の2つの関係を表す．図1のPascalのソースプログラムについてのプログラム依存グラフは図2のようになる．

プログラム中の文sの変数vに関する静的スライス

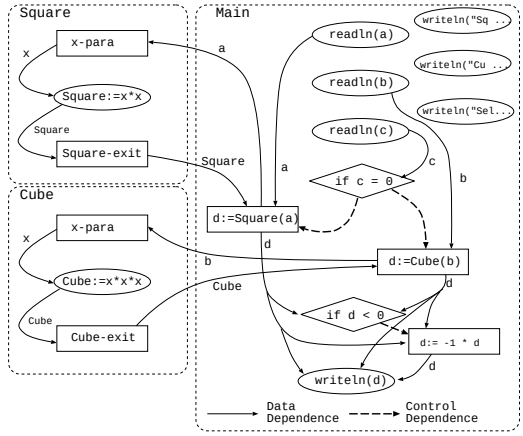

図2　プログラム依存グラフ
Fig. 2　Program Dependence Graph (PDG)

(組 (v, s) をスライシング基準と呼び，どの文に関してスライスを計算するのかを表す．) は，スライシング基準に対応する節点から 制御依存辺，データ依存辺を逆に辿って到達できる節点に対応する文の集合である(注1)．例えば，図1のプログラムの文24の変数 d をスライシング基準として静的スライスを計算すると，図3のようにwrite文(12，14，16行)を除いたすべての文が含まれる．

2.2　動的スライス

静的スライスの計算はソースコードを対象として，依存関係を計算し，スライスを抽出していたが，動的スライスの計算では，依存関係を計算する対象は実行系列である．実行系列とは，ある入力を与えプログラムを実行したときの，実際に実行された文の列をいう．また，実行系列中の p 番目の文の実行のことを実行時点 p と呼ぶ．

実行系列 e 中の実行時点 r_1，r_2 について考える．次の条件をすべて満たしているとき，実行時点 r_1 から実行時点 r_2 への動的制御依存 *(Dynamic Control Dependence，DCD)* があるという．

- 実行時点 r_1 が条件文である．
- 実行時点 r_2 が実行されるかどうかは，実行時点 r_1 の結果に依存する．

次の条件をすべて満たすとき，実行時点 r_1 から実行時点 r_2 へ変数 v に関して動的データ依存 *(Dynamic Data Dependence，DDD)* があるという．

(注1): データ依存辺を辿るには，まず v に関する辺を辿り，以降，影響を与える変数の辺のみを推移的に辿る．

```
 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
13    readln(a);
14
15    readln(b);
16
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3　24行目変数 d に関する静的スライス
Fig. 3　Static Slicing Result by d at Line 24

- 実行時点 r_1 で変数 v を定義している．
- 実行時点 r_2 で変数 v を参照している．
- 実行時点 r_1 から r_2 の間に変数 v を定義している実行時点が存在しない．

動的スライスのスライシング基準 (x, r, v) は，入力 x，実行時点 r，変数 v の3つからなる．スライシング基準 (x, r, v) に対応する動的スライスとは，r から動的制御依存，動的データ依存関係を逆向きにたどることによって得られた到達可能な文の集合である．

図1のプログラムについてのスライシング基準が入力 $(a = 2\ b = 3\ c = 0)$，実行時点13(24行目の文を実行した時点)，変数 d である動的スライスの計算結果を図4に示す．

```
 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4 入力データ $(a = 2, b = 3, c = 0)$ 24行目変数 d に関する動的スライス
Fig. 4 Dynamic Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$

2.3 静的スライスと動的スライスの特徴

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を構築する際に，制御依存関係はプログラムの構造を調べることで容易に求めることができる．また，データ依存関係はデータフロー方程式[3]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，現実には短い時間で計算できる[15], [17]．しかし，静的スライスはすべての実行可能なパスについて考えているため，多くの場合，静的スライスの計算結果に，元のプログラムの大部分が含まれてしまい，参照範囲を小さくできない．

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．また，特定のテストデータに対して不具合が発生した場合，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，フォールト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，フォールト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の位置特定を行う場合に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．このために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要することがある．

3. コールマークスライス

3.1 概　　要

本論文で提案するコールマークスライスは，静的スライスと動的スライスの欠点を解決するために静的な情報と動的な情報を共に用いる．前節で述べた動的スライスの問題は次の2つにまとめられる．

- 実行系列の記録と動的な依存関係の解析
- 動的な依存関係からのスライスの計算

前者に関し，我々は正確な情報のかわりに，関数または手続き呼出し文が実行されたかどうかの情報のみを記録することとした．後者に関して，我々は動的な依存関係ではなく静的な依存関係を解析することで解決する．

コールマークスライスは，ある入力データを与えてプログラムを実行したときに，静的スライスからその実行において影響を与えなかった文のうち，その影響を与えなかった旨の事実を以下の方法で計算できた文を取り除いて得られる文の集合からなる．本手法では確実に影響を与えない文を計算するため，プログラムのある部分においてある機能を実行したかどうか，つ

まり，関数呼出し文が実行されたかどうかに注目する．

3.2 諸定義 (ED，CED)

コールマークスライスを計算するために，新たな文間の依存関係を定義する．次の条件を満たすとき，文 s_1 から文 s_2 へ実行依存関係 *(Execution Dependence，ED)* があるという．

- 文 s_2 が実行されなかった場合， 文 s_1 が必ず実行されない．

プログラム中のすべての実行依存関係を調べるには動的な情報が必要であるが，その部分集合が静的解析により求めることができる．すなわち，s_1 と s_2 が制御フローグラフ (CFG) [3] の同じ基本ブロック内にあるとき，つまり 2 文の間に出ていくパスと入って来るパスがないとき，s_1 と s_2 は互いに実行依存関係にある．また，s_2 が s_1 の基本ブロックを支配する基本ブロックであるとき，s_1 は s_2 に実行依存している．基本ブロックに関する制御フローや支配関係は静的な解析により簡単に求められる [3]．

次に文 s に対し支配呼出し文集合 $CED(s)$ を以下のように定義する．

$$CED(s) \equiv \{t \mid t \text{は関数または手続き呼出し文 かつ } s \text{は} t \text{に実行依存している}\}$$

文 s の実行は $CED(s)$ に含まれる呼出し文に支配されている．$CED(s)$ に含まれる呼出し文のどれか一つでも実行されていない場合，s は実行されていないと決定できる．次の簡単なプログラムで CED の例を示す．

```
     ...
s1: call A ;
s2: if a=1 then begin
s3:     b:= c ;
s4:     call B ;
     ...
```

これで，$s1$ と $s2$ は互いに実行依存している． また $s3$ と $s4$ も互いに実行依存している． 更に $s3$ と $s4$ は，それぞれ $s1$ 並びに $s2$ に実行依存している．この結果， $CED(s2) = \{s1\}$， $CED(s3) = \{s1, s4\}$ が得られる．

3.3 入 力 言 語

本論文では，静的スライス及び動的スライスとの比較を行うために，これまでに開発しているスライスシステムに実装した [15] (4.1節にて述べる)．そのためにスライスシステムの入力言語をコールマークスライスにおいても入力言語としている．入力言語は以下のようなものである．この言語には文として条件文 (if 文)，代入文，繰返し文 (while 文)，入力文 (readln 文)，出力文 (writeln 文)，手続き呼出し文，複合文 (begin–end) がある．変数の型としてはスカラ型のみを考える．プログラムは，大域変数宣言，手続き (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及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，自己再帰的及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．

本論文では上記のような言語を入力言語としているが，ポインタ変数や goto 文のような非構造な文を含む言語でも，実行依存関係は計算可能であるので，このような言語に対してもコールマークスライスは適用可能である．

3.4 コールマークスライスの定義

まず，コールマークスライスの直感的な定義を与える．プログラム P のある入力 x に対する実行 e に関して，自分の支配呼出し文集合中のすべてが実行されている文の集合 S_1 を考える．P のスライシング基準 (s, v) に関する静的スライスを S_2 としたとき，$S_1 \cap S_2$ を「スライシング基準 (x, s, v) に関するコールマークスライス」と呼ぶ．静的スライスに含まれているが，コールマークスライスには含まれないような文は実行 e において実行されなかったか，または，実行されていない文とのみ実行依存関係があることを意味している．

コールマークスライスは，形式的に次の Step 1 ~ Step 3 で定義される．

Step 1　実行前の依存関係解析

静的スライスの計算と同様に， データ依存関係と制御依存関係を解析し，PDG を構築する．

Step 2　実行時の記録

プログラムを実行し， 関数または手続き呼出し文が実行されるごとに， その呼び出し文に対応する PDG 上の節点に “executed” と記録する． この記録された呼び出し文の集合を CM と呼ぶ．

Step 3　コールマークスライスの抽出

図 5に示すアルゴリズムでコールマークスライスを抽出する．

図 1のプログラムについてのスライシング基準 $(\text{line24}, d)$ に関する静的スライスは図 3のようになる．このプログラムに入力 $(a = 2, b = 3, c = 0)$

Inputs
PDG: Program Dependence Graph
CM: 実行されたcall文の集合
(s_c, v): スライシング基準．s_cは文，vは変数名．
Temporary
M ,N: 節点の集合
m ,n: 節点
Output
M: コールマークスライスとして得られる節点の集合
Algorithm Body
(1) $M \leftarrow s_c$
(2) $N \leftarrow \{n \mid n \xrightarrow{v} s_c\} \cup \{m \mid m \dashrightarrow s_c\}$
(3) While $N \neq \phi$ である間以下の動作を繰り返す．
(a) $n \in N$ を1つ選ぶ
(b) $N \leftarrow N - n$
(c) if $CED(n) \not\subseteq CM$ であるとき (a) に戻る
(d) $M \leftarrow M \cup n$
(e) $N \leftarrow N \cup \{m \mid m \notin M \wedge (m \xrightarrow{w} n \vee m \dashrightarrow n)\}$
ここでwは文nで参照する各変数名

図5 コールマークスライス抽出のアルゴリズム
Fig. 5 Algorithm of Post-Execution Collection for Call-Mark Slicing

を与えて実行した場合について考える．このとき，$CM = \{19\}$となり，静的スライスと同様のスライシング基準に関するコールマークスライスは図6のようになる．

4. コールマークスライスの実装

4.1 スライスシステムの概要

コールマークスライスの評価を行うために，我々のグループが開発したPascalプログラムのスライスシステム[15]にコールマークスライスの機能を追加した(図7)．このシステムでは，ソースプログラムは抽象構文木(abstracted source program)に解析され，記憶される．ユーザは対話的で視覚的なエディタによりソースプログラムの参照，変更が可能である．ソースプログラムはユーザの入力により解析されPDGに変換される．静的スライスは指定されたスライシング基準によりPDG上で計算される．ソースプログラムや静的スライスは，インタープリタ(executor)により実行することができる．デバッガは，トレース，ブレークポイントの設定などの機能がある．実行時には動的な変数の依存関係を記憶し，動的スライスの計算に用いる．システム全体のサイズは，コールマークスライスに関する部分も含めて，C言語で約19，000行である．コールマークスライスの実装は前章で述べた方法に基づいて行った．

```
 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    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

図6 コールマークスライスの結果(スライシング基準：24行目の変数d, 入力データ：$(a = 2, b = 3, c = 0)$)
Fig. 6 Call-Mark Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$

4.2 プログラムの実行

このシステムを用いて，いくつかのプログラムを実行した．表1に3種類のプログラム(P1:カレンダーの出力，P2:酒屋問題[20]，P3:拡張酒屋問題)の実測データを示す．これらの値はスライシング基準や入力データによって異なるが，今回の実験では典型的なデバッグの状況を想定し，これらを選んだ(多くの場合，スライシング基準はプログラムの終りにある出力変数である)．

表2に，実行前に必要な解析時間を示す．静的スライスにおいてはPDGの構築に必要な時間，コールマークスライスではPDGの構築とCEDの計算にかかる時間を表している．動的スライスではこの段階での解析は必要ない．

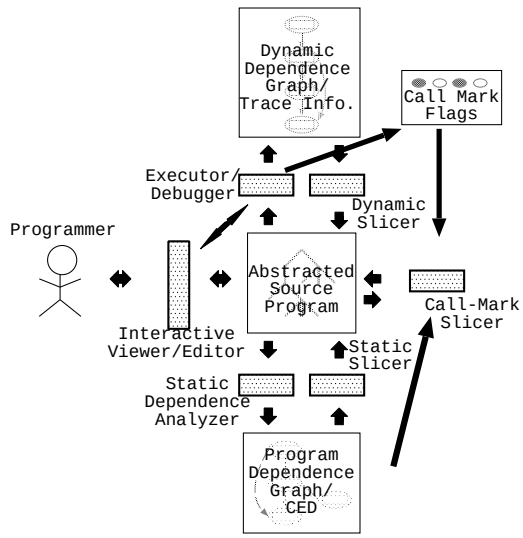

図7　スライスシステムの構成
Fig. 7　Architecture of Slicing System

表3に，プログラムの実行にかかった時間を示す．静的スライスの計算には必要ないが元のプログラムの実行にかかる時間として他との比較のために記載している．動的スライスの実行時には，変数間の動的な依存関係の解析も同時に行うためその時間も含まれる．コールマークスライスでは実行時間に関数呼び出し文の記憶に必要な時間が含まれる．

表4に，スライスの抽出にかかった時間を示す．静的スライスはプログラム依存グラフ上の依存関係を辿って計算するのに必要な時間，動的スライスでは動的な依存関係を辿って計算するのに必要な時間．コールマークスライスではStep 3 に必要な時間である．

表1　抽出されるスライスのサイズ(行)
Table 1　Size of Various Slicing Results (lines of code)

program	static	dynamic	call-mark
$P1$ (88 lines)	27	14	22
$P2$ (387 lines)	175	139	156
$P3$ (941 lines)	324	50	166

表2　実行前の解析時間(ミリ秒)
Table 2　Pre-Execution Analysis Time(ms)
(Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	22	N/A	23
$P2$	1,275	N/A	1,362
$P3$	5,652	N/A	8,670

表3　実行時間(ミリ秒)
Table 3　4 Execution Time(ms)
(Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	38	87	47
$P2$	48	903	53
$P3$	4,046	31,635	4,104

表4　スライス抽出にかかる時間(ミリ秒)
Table 4　Slice Collection Time(ms)
(Pentium-II 300MHz with 256MB Memory)

program	static	dynamic	call-mark
$P1$	1	199	1
$P2$	5	2,863	8
$P3$	93	1,182	80

5. 考　　察

5.1　プログラム実行時間

- 抽出されるスライスサイズ

表1に示すように，コールマークスライスのサイズは，静的スライスと動的スライスの中間の値になった．

コールマークスライスは，以下の条件を満たす文を静的スライスから取り除いたものである．

条件1　CMとCEDを元に実行されなかったと判定可能な文

条件2　実行されなかったと判定された文のみと依存関係のある文

よって，コールマークスライスは同じスライシング基準に対する静的スライスの部分集合になる．

動的スライスは上記の2つの条件を満たす文は含まない．また，コールマークスライスは実際には実行されていないが，CED及びCM を用いたとしても，取り除くことのできない文が存在する．例えば，ある文Sがある条件文により実行が支配されるブロック内に存在し，そのブロック内及びそれを支配する上位ブロック内に関数または手続き呼び出し文が存在しないとすると，$CED(S) = \phi$であるために，そのブロックが実行されたかどうかが判定できず，そのブロックはコールマークスライスに含まれてしまう．よって，動的スライスはコールマークスライスの部分集合になる．

- 実行前の解析時間

表2で示すように，コールマークスライスでは静的スライスと比較して少し余分な時間が必要である．これは，PDGの構築に加えて，実行依存関係の解析が必要であるからである．

- 実行時間

表3で示すように，動的スライスのオーバーヘッドがかなり大きい．$P3$の実行では，90Mの実メモリを使用している．もしプログラムの実行が長くなれば，このオーバヘッドがさらに増す．一方，コールマークスライスでは，静的スライスの計算における実行(元のプログラムの実行)と比較してオーバーヘッドの増加が少ない．これは呼出し文の記録に必要な時間が非常に小さいことを示している．

- スライス抽出時間

表4で示すように，動的スライスの抽出には長い時間がかかる．コールマークスライスの時間は静的スライスの時間と同程度である．またプログラム$P3$においては，静的スライスにおいて必要な時間よりも小さい．これは，コールマークスライスはPDGから実行に関係ない部分を取り除くために，辿るPDGの範囲が静的スライスよりも小さいからである．

5.2 関連研究

コールマークスライスはプログラマの参照範囲を限定するための効率的で実用的な方法である．実行前の解析時間，実行時間，スライス抽出にかかる時間は静的スライスと同程度で，スライスの有効性は静的スライスよりも大きく，この方法は有効性と効率性の折り合いがとれていると思われる．これまでに，コールマークスライス以外にも，静的な情報と動的な情報を混合するような関連する研究が行われている[4], [6]．

文献[1]では，動的な情報を用いたスライスの抽出方法として静的な依存グラフからスライスを求める手法について述べている．しかし，ここで述べられている手法では各文が実行されたどうかの情報を持つ必要があるために，コールマークスライスと比べて多くの記憶領域と情報を記憶するための実行のオーバーヘッドを要する．また，この手法で計算されるスライスは実行時に各文の情報を記憶する必要があるにもかかわらず，動的スライスと比較すると余分な文がスライスに含まれる結果になる[12]．

混合スライス(Hybrid Slice)[6]は，ブレークポイントと関数の呼び出し履歴の情報を用いて静的スライスの改良を行っている．前者はプログラマによって与えられ，実行された制御フローの推定に用いる．後者は，関数の呼び出しと戻る点の間の動的スライスを計算するために用いる．この方法は，我々の方法よりも動的な情報を多く用いるため，我々の方法よりも動的スライスに近くなるが，スライスの結果を良くするためには，適切な位置にブレークポイントを設定しなければならない．一方，我々の方法は入力データを与えることと，スライシング基準を指定すること以外は，自動的に計算することができる．また，混合スライスは関数呼び出し履歴(calling history)を記憶するのために，非常に多くの記憶領域が必要となる．この必要な領域は実行系列の長さに依存する．しかし我々の手法では呼出し文の実行の記憶に必要な領域だけであり，これはプログラムのサイズにほぼ比例する．プログラムの実行が長い場合に，必要な記憶領域の違いが明らかになる．

制約スライス(constrained slice)[4]は，静的スライスと動的スライスを一般化したもので，プログラムの記号実行手法を用いている．入力として制約条件を与える．この入力の制約を用いて，プログラムを書き直し，依存関係を解析する．この方法は，静的スライスと動的スライスの一般化また部分評価とプログラムの簡易化を含んでおり，面白いが，このような一般化の方法が効率的に実装されたり，実際に有効であるということは知られていない．

6. まとめ

作業者の注目する範囲を限定することは，デバッグや保守の効率の改善に非常に重要であるが，しかし，従来のプログラムスライス技法には有効性や効率の面で問題がある．そこで，本論文では効率が良く効果的なプログラムスライス技法として，コールマークスライスを提案した．この技法は静的スライスと同程度の時間で依存関係の解析，実行が可能である．また得られるスライスのサイズは，静的スライスよりも小さく動的スライスよりも大きくなる．我々は，実際にこのスライスの抽出アルゴリズムを実装し，この技法の評価を行った．

現システムでは，静的依存解析(PDG構築)をプログラムを実行する前に行っているが，今後，静的依存解析をプログラム実行後(コールマーク情報を得た後)に行うことを検討している．これにより，実行に関係のない部分の依存関係の計算を省くことができ，依存解析時間をより小さくすることができる．

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, R. A., and Spafford, E. H.: "Debugging with Dynamic Slicing and Backtracking", *Software Practice and Experience*, Vol. 23,

No. 6, pp. 589–616, 1993.

[3] Aho, A. V., Sethi, R., and Ullman, J. D.: “Compilers: Principles, Techniques, and Tools”, *Addison Wesley*, Massachusetts, 1986.

[4] Field, J., and Ramalingam, G.: “Parametric Program Slicing”, *Proc. of 22nd ACM Symposium on Principles of Programming Languages*, pp. 379–392, San Francisco, USA, January. 1995.

[5] Atkinson, D. C. and Griswold, W. G. : “The Design of Whole-Program Analysis Tools”, *Proceedings of the 18th International Conference on Software Engineering*, pp. 16–27, Berlin, Germany, March. 1996.

[6] Gupta,R., and Soffa, M. L. : “Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information”, *Proceedings of the 3rd International Symposium on the Foundation of Software Engineering*, pp. 29–40, October. 1995.

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

[8] 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.

[9] Korel, B., and Laski, J. : “Dynamic Program Slicing”, *Information Processing Letters*, Vol.29, No,10, pp. 155–163, 1988.

[10] Korel, B., and Laski, J. : “Dynamic Slicing of Computer Programs”, *Journal of Systems Software*, Vol.13, pp. 187–195, 1990.

[11] Murphy, G. C., and Notkin, D.: “Lightweight Lexical Source Model Extraction”, *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, pp. 262–292, July. 1996.

[12] 直井，高橋: “経路依存フローグラフを用いたプログラムスライシング”, 信学論, Vol. J78-D-I, No. 7, pp. 607-621, 1995.

[13] Ning, J. Q., Engberts, A., Kozaczynski, W. V. : “Automated Support for Legacy Code Understanding”, *Communications of the ACM*, Vol. 37, No. 5, pp.50–57, May. 1994.

[14] 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，SS98-3，May. 1998.

[15] 佐藤, 飯田, 井上: “プログラムの依存関係解析に基づくデバッグ支援システムの試作”, 情処学論, Vol. 37, No. 4, pp. 536–545, 1996.

[16] 下村 隆夫: “プログラムスライシング技術と応用”, 共立出版，1995.

[17] 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”, 信学論, Vol. J78-D-I, No. 1, pp. 11–22, 1995.

[18] Vengatesh, G. A., and Fischer, C. N. : “SPARE: A Development Environment for Program Analysis Algorithms”, *IEEE Trans. on Software Engineering*, Vol. 18, No. 4, pp.304–315, April. 1992.

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

[20] 山崎利治:“共通問題によるプログラム設計技法解説”，情報処理学会誌，Vol. 25, No. 9, p.934, 1984.

（平成年月日受付，月日再受付）

西松　顯

平8阪大・基礎工・情報卒．現在同大学院修士課程在学中．プログラムスライスの研究に従事．

地平　稔

平9阪大・基礎工・情報卒．現在奈良先端科学技術大学院大学修士課程在学中．プログラムスライス，ユーザインターフェースの研究に従事．

楠本　真二　（正員）

昭63阪大・基礎工・情報卒．平3同大学院博士課程中退．同年同大・基礎工・情報・助手．平8同大講師．工博．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．情報処理学会，ＩＥＥＥ各会員．

井上　克郎　（正員）

昭54阪大・基礎工・情報卒．昭59同大大学院博士課程了．同年同大・基礎工・情報・助手．昭59～昭61ハワイ大マノア校・情報工学科・助教授．平1阪大・基礎工・情報・講師．平3同学科・助教授．平7同学科・教授．工博．ソフトウェア工学の研究に従事．

Fig.1　ソースプログラム
Pascal Source Program
Fig.2　プログラム依存グラフ
Program Dependence Graph (PDG)
Fig.3　24行目変数dに関する静的スライス
Static Slicing Result by d at Line 24
Fig.4　入力データ$(a = 2, b = 3, c = 0)$ 24行目変数dに関する動的スライス
Dynamic Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$
Fig.5　コールマークスライス抽出のアルゴリズム
Algorithm of Post-Execution Collection for Call-Mark Slicing
Fig.6　コールマークスライスの結果(スライシング基準:24行目の変数d, 入力データ:$(a = 2, b = 3, c = 0)$)
Call-Mark Slicing Result by d at Line 24 with input $(a = 2, b = 3, c = 0)$
Fig.7　スライスシステムの構成
Architecture of Slicing System
Table 1　抽出されるスライスのサイズ(行)
Size of Various Slicing Results (lines of code)
Table 2　実行前の解析時間(ミリ秒)
Pre-Execution Analysis Time(ms)
Table 3　実行時間(ミリ秒)
4 Execution Time(ms)
Table 4　スライス抽出にかかる時間(ミリ秒)
Slice Collection Time(ms)