プログラムの依存関係解析に基づくデバッグ支援ツールの試作

佐藤慎一[†] 飯田元[††] 井上克郎[†]

ソースプログラムの依存関係解析に基づき，スライスの抽出や部分評価を行い，デバッグの対象となる部分を小さくすることができるデバッグ支援ツールを試作した．本ツールでは，ステップ実行やトレースなどを元のプログラムだけでなく限定されたプログラム部分に対しても行うことができる．本ツールを用いれば，プログラム中でバグに関係のある部分のみを作業の対象とすることができるため，効率の良いデバッグを行うことができる．

Software Debug Supporting Tool Based on Program Dependence Analysis

SHINICHI SATO,[†] HAJIMU IIDA[††] and KATSURO INOUE[†]

Debugging large and complex software is not easy task since localizing and identifying faults are very diffcult. We have depeloped a debug tool which extracts program slices and which partially evaluates program texts based on program dependence analysis. Using this tool, we can extract parts of the program which would relates to bugs, and also we can execute and trace the extracted parts. Using this tool, we would expect that the debugging process will be more efficient.

1. まえがき

年々プログラムは多機能化し，大規模になっている．そのような大規模なプログラムをデバッグするのは容易ではない．

通常，プログラムをデバッグする際には，対象となるプログラム全体を読んで理解する必要がある．したがって，プログラムが大きくなるとプログラム全体の把握やエラーに関する部分の特定に多くの時間がかかってしまう．

もし，エラーに関係があると思われる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，開発者はプログラムの一部分だけに注目してエラー位置の特定ができ，開発効率の向上が期待される．ただし，その際に開発者がエラーに関連のあるプログラムの部分の抽出を手動で行うのは手間がかかるうえ，ときとして関連のある部分を見過ごす危険がある．したがって，そのようなプログラムの部分を機械的に効率良く抽出する方法が必要である．

本研究では，プログラムスライス（以降，単にスライスと呼ぶ）および部分評価手法を用いて，プログラム全体から注目する文や変数に関係のある部分のみを抽出し，それに対して処理を行うようなデバッグ支援ツールを提案し，実際にツールの試作を行った．

スライスとは，直観的には，プログラム中のある文 s での変数の値に影響を与える文の集合，もしくは s での変数の定義が影響を与える文の集合である．スライスを用いることにより，ある出力変数に関係のある文のみをプログラムから抽出することができる．

また，部分評価とは，プログラムに対する入力変数の値をある定数に固定し，プログラム中の文や式の書換えを行い，目的とする部分の抽出を行うことをいう．

本研究では，動的スライス（プログラムの実行系列を解析して得られるスライス）ではなく，静的スライス（プログラムの依存関係を解析し，それに基づいて求められるスライス）を利用することにより，抽出したスライスを1つのプログラムとして扱ってデバッグを行うことができる．

さらに，スライスによって有効な抽出が行いにくい場合でも本ツールでは，部分評価の機能を持つため，対象を小さくできる場合がある．たとえば入力のパラメータでサブシステムの1つの機能の実行が指定/除

† 大阪大学 基礎工学部 情報工学科
Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University
†† 奈良先端科学技術大学院大学 情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology

外されるようなプログラムに対しては，入力パラメータを特定の値として部分評価することによって，容易にそのサブシステムを抽出したり除いたりすることができる．

本ツールにおいて背景となるスライスや部分評価の手法は，それぞれ独立して詳しく研究されている[6),15)]．しかし，このような静的スライスと部分評価を組み合わせたデバッグ支援ツールは今までに存在しない．

ここでは，比較的実現が容易でかつ実行効率が現実的なスライスや部分評価の方法を用いて試作ツールを作成し，この種のツールの有効性を調べる．スライスについては，(再帰/非再帰) 関数間の変数依存解析[17)]を実現している．ここで実現されている方法に加え，ポインタ変数の解析[2),10)]や，それにともなう変数のエイリアス解析[11)]，なども考えられるが，それらを正確に扱えるアルゴリズムがまだ存在しないことから，その実行効率やその実用性が不明であるため，本試作ツールでは実現しなかった．部分評価は，値の代入による式の評価と不要式の削除という方法に基づいている[16)]．プログラムの記号実行やループの展開[3)]，ブレスブルガ式の評価[8)] などの技法も採用できると考えられるが，実現の困難さや非効率さ，さらに，デバッガとして用いるには元のプログラムとの対応関係を保存する必要性があることなどから本ツールではより簡単な方法を用いた．

本ツールを用いることにより，デバッグ時に，バグに関係のある部分のみをスライスもしくは部分評価で抽出して，その部分のみを実行させることが可能となり，その結果，デバッグの対象となるプログラムの参照部分を減少させることができた．

これまでにも，スライスを利用したデバッグ支援ツールはいくつか発表されている[1),14)]．しかし，これらはいずれも動的スライスに基づくもので，実行させるたびにスライスを計算しなければならないうえ，実行系列を保存しておく必要があるために解析結果が膨大な量になる．

また，静的スライスを用いたデバッグ手法としては，ダイシング[9)]やアルゴリズミック・デバッギング[4),5)]があるが，本研究では，スライスや部分評価を行うことにより，プログラムのうち，一部の機能のみを実行する部分を抽出し，その抽出部分をデバッグの対象とすることによりデバッグ時の参照範囲を縮小させることを目的としている．一方，ダイシングやアルゴリズミック・デバッギングは，いきなりバグ位置を特定しようとしている点が，本研究とは異なる．本ツールは，抽出した部分を1つのプログラムとして扱うことができるため，必要ならば，それらの手法を抽出部分に適用することもできる．

本稿では，これ以降，2章でプログラムの依存関係解析の概要を，3章で作成したツールの概要を，4章でツールの実行例およびその評価について述べる．

2. プログラムの依存関係解析

本稿で述べるデバッグ支援ツールは，プログラムの依存関係解析の結果得られるプログラム依存グラフ (Program Dependence Graph，略して PDG) から，スライシングや部分評価を行うことによってプログラムの参照範囲を小さくすることによりデバッグの支援を行う．

2.1 プログラム依存グラフ (PDG)

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および 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) 関係および条件文や繰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，

- プログラム中の文 s で変数 v を定義している
- プログラム中の2つの文 s から t へのすべての実行パスの中で，v を定義しないパスが少なくとも1つ存在する

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 を参照するとき，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とき，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おり，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われる．詳細は，文献

17) を参照されたい.

例として図1のプログラムに対応するPDGを図2に示す. 図のPDGの中で角の丸い四角がプログラム中の各文に対応する節点で, 楕円が中継節点である. また, 有向辺のうち, 実線で名前がついているものがDD関係の辺で, 破線のものがCD関係の辺である. 実線についている名前は, そのDD関係の辺が影響を伝える変数の名前である. また, 破線で囲まれている部分はプログラム上の1つの手続きを表す.

2.2 スライス

1章で述べたように, 本ツールで扱うスライスは, 静的スライスである. 文 s における変数 v に関するスライスとは, PDG上において, CD関係の辺またはDD関係の辺をたどって文 s の変数 v に到達できる節点集合に対応する文の集合である[17].

PDGを与えられた頂点から辺の順方向に探索していくものをforward sliceと呼び, 逆方向に探索していくものをbackward sliceと呼ぶ.

スライスの例を図1に述べる. このプログラムの36行目で参照されている変数gに関するbackward sliceが, 図中の下線部分に示されている. この場合, 変数gに影響を及ぼさない部分, すなわち関数lcmはスライスに含まれていない.

2.3 部分評価

本稿でいう部分評価とは, プログラムに対する入力変数のとる値を固定し, それを参照する式や変数の値を順次書き換えたり, 実行されない不要な文や手続きの削除を行うことをいう. 部分評価を適用すれば, 入力変数, もしくはそれに相当する大域変数の値によってどの機能を使うか, もしくは使わないかを決定しているようなプログラムに対しては, 使用しない機能に関する部分をプログラムから削除することができる.

部分評価手法の詳細は文献16) に述べられているので, 本稿では, あるプログラム P の部分評価手法の概略を示す.

(1) P の大域変数の中から固定の対象とする入力変数 x (複数個でもよい) を選ぶ.

(2) 目的とする機能を選び出すのに応じて x の値 (特定の値 a) を定め, 変数とその値の組 $\langle x, a\rangle$ を作る. そのほかの大域変数と局所変数 y については不定値 $\perp$ との組 $\langle y, \perp\rangle$ を作る. (各変数の組の集合を状態と呼び, 特にこのような値の割当てを初期状態 S という.)

(3) 主プログラムの本体 M および初期状態状態 S を引数として関数 *simple* (M, S) を呼び, その返り値を部分評価の結果として出力する.

```
 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;
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 PDGの元のプログラム
Fig. 1 Original program of PDG.

ここで, *simple* は与えられた状態の下で, 入力文を部分評価した結果の文を返す. *simple* は文の種類ごとに定義されており, (右辺の値が決まる場合の) 代入文の実行, (条件式の値が決まる場合の) 条件分岐文の書き換え, (同じく条件式の値が決まる場合の) 繰返し文の書き換えを行う. また, いくつかの文の系列であ

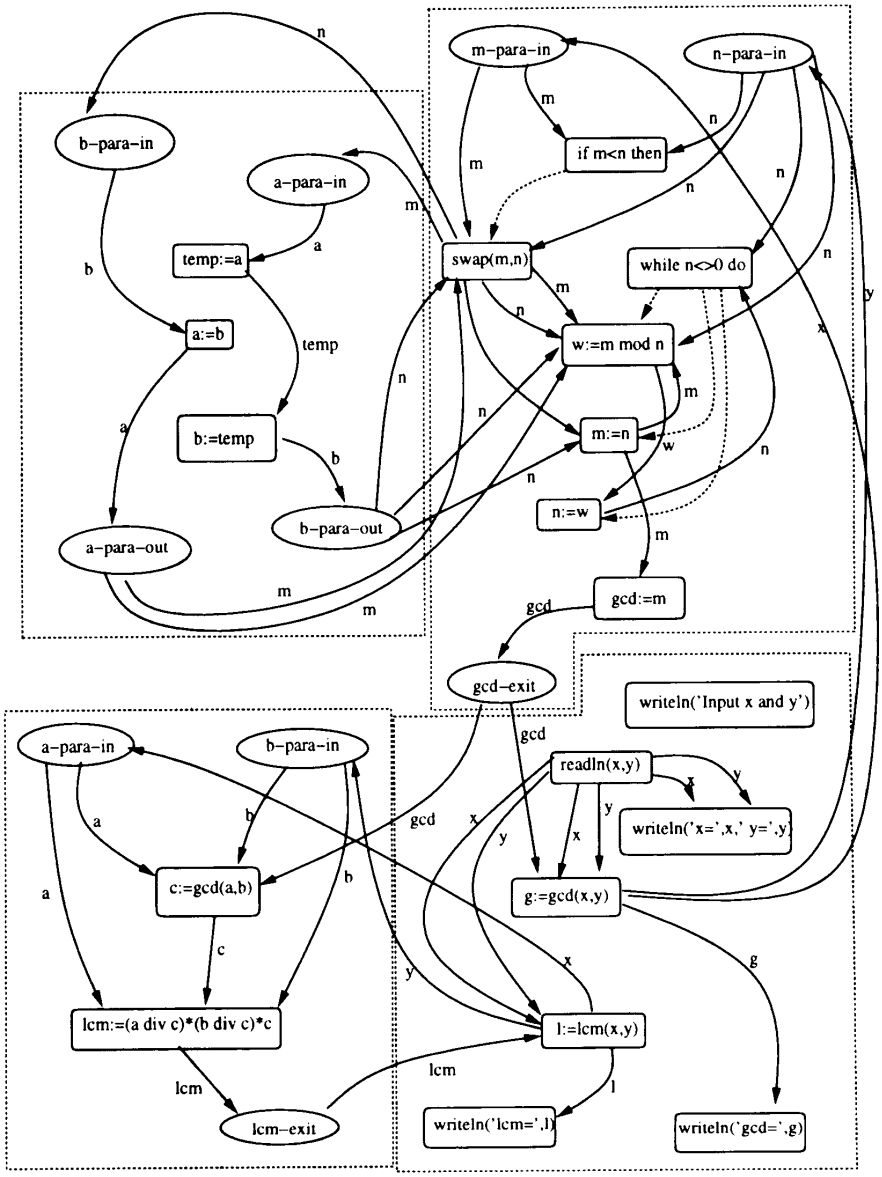

図2 PDG の例
Fig. 2 The PDG.

る複合文は，前の文から順に再帰的に *simple* を実行することにより部分評価される．詳細は文献 12), 16) を参照されたい．

上記の手順でプログラムが部分評価されると，元のプログラムにあった手続きの呼び出しや，変数の定義，参照がなくなることが起こりうる．手続きが一度も呼ばれなかったり変数が一度も参照されない場合，それらの手続きの定義や変数宣言が不要となり，その定義や宣言文を削除できる．これらはプログラムを解析した結果生成される PDG からその依存関係を調べることによって容易に調べることができる．

部分評価の例として，**図3** のプログラムの入力変数のうち，変数 b の値を 'n' に，変数 c の値を 'y' に固定して部分評価を行った結果のプログラムを**図4** に示す．

入力変数 b の値を 'n' に固定した結果，入力文が削除され，また，その値が 'y' のときに実行されるはずであった 27 行目から 37 行目の部分が削除されている．また，入力変数 c の値を 'y' に固定した結果，24 行目の if 節が削除されている．

3. ツールの概要

本章では，本研究で試作したデバッグ支援ツールの概要について述べる．本ツールの実装には，解析や実行部には C 言語を，インタフェースには Tcl/Tk を用いた．

```
1  program wordcount(in,out);
2    var a,b,c:char;
3        in:array[0..1000] of char;
4        i:integer;
5        letter,word,line:integer;
6        isinword:boolean;
7
8
9    begin
10     writeln('Count the letter?(y/n)');
11     readln(a);
12     writeln('Count the word?(y/n)');
13     readln(b);
14     writeln('Count the line?(y/n)');
15     readln(c);
16     i := 0;
17     letter := 0;
18     word := 0;
19     line := 0;
20     isinword := false;
21     while in[i] <> EOF do begin
22       if a = 'y' then
23         letter := letter + 1;
24       if c = 'y' then
25         if in[i] = EOL then
26           line := line + 1;
27       if b = 'y' then
28         if (in[i] <>' ')and(in[i] <> EOL) then
29           begin
30             if isinword = false then
31              begin
32               isinword := true;
33               word := word +1
34             end
35           end
36         else
37           isinword := false;
38       i := i +1
39     end;
40     if a = 'y' then
41       writeln('letter =',letter);
42     if b = 'y' then
43       writeln('word   =',word);
44     if c = 'y' then
45       writeln('line   =',line)
46   end.
```

図3 部分解釈前のプログラム
Fig. 3 Original program before partial evaluation.

3.1 言語仕様

本ツールが対象とする言語は，以下のような仕様を持つ Pascal のサブセットである．

- 文として，代入文，条件文，繰返し文，手続き呼出文，begin–end で囲まれる複合文を扱う．
- 手続きは再帰呼び出しも扱う．ただし，部分評価ではこれは扱えない．手続きの引数の渡し方については，値渡しと変数渡しの 2 種類がある．

```
program wordcount(in,out);
  var a,b,c:char;
      in:array[0..1000] of char;
      i:integer;
      letter,word,line:integer;
      isinword:boolean;

  begin
     writeln('Count the letter?(y/n)');
     readln(a);
     writeln('Count the word?(y/n)');
     writeln('Count the line?(y/n)');
     i:=0;
     letter:=0;
     line:=0;
     while in[i] <> EOF do begin
        if a='y' then

           letter:=letter+1;
        if in[i]=EOLN then
           line:=line+1;
        i:=i+1
     end;
     if a='y' then

        writeln('letter =',letter);
     writeln('line =',line)
  end.
```

図4 部分解釈後のプログラム
Fig. 4 Result of partial evaluation.

- 変数のデータ型はスカラー型のみでポインタ型は扱わない．具体的には整数型，文字型，論理型およびそれらを要素に持つ配列型とした．

このように，簡単化されてはいるものの，プログラミング言語の基本的な文の構造はすべて含んでいるので，拡張は容易に行うことができる．

3.2 機　　能

本ツールは以下のような特長を持つ．

- 通常のデバッガと同様にプログラムのステップ実行，変数の参照，ブレークポイントの設定などが行える．
- プログラムスライスとして backward slice，forward slice のどちらでも抽出でき，また，そのときに，PDG のうち制御依存とデータ依存の片方だけを探索したり，希望の段階までを探索することもできる．
- 入力変数もしくは入力に相当する変数の値を定数に固定することにより，部分解釈を行って，プログラムの一部の機能のみを実現する部分のみを抽出することができる．
- 抽出したプログラム部分に対してもデバッグ作業を行える．

- プログラムの抽出部分を1つのプログラムとして構文的に正しくなるように修正し，保存できる．

3.3 ツール構成およびその動作

本ツールの構成図を**図5**に示す．本ツールは，ソースコードの構文解析を行いつつプログラム中の各文をPDGの各節点と対応付けるパーザ部，その結果に基づいてPDGを作成するアナライザ部，PDGを探索してスライスを得るスライサ部，PDGを利用してプログラムの部分評価を行う部分評価部，プログラムの実行を行うインタープリタ部に分けられる．

ユーザがデバッグの対象となるプログラムを入力すると，ツールは，パーザ部で構文解析を行った後，アナライザ部で依存関係の解析を行う．それらの解析が終了すると，PDGおよび実行に必要な構文木や変数の情報などが生成される．

ユーザはこの段階でツールにどの動作をさせるかを指定する．ユーザがプログラムを実行させた場合には，ツール内では，インタープリタ部にアナライザ部が生成した構文木や変数の情報などが渡され，プログラムが実行される．その際，ユーザは，通常の実行だけでなく，ステップ実行やブレークポイントの設定，変数値の参照などのデバッグに必要な作業も行うことができる．

部分評価を行う場合，ユーザは，まず，どの入力変数をどの値に固定するのかといった情報を入力する．その後，ツール内では，その入力制限情報とPDGおよび構文木などの情報が部分評価部に渡される．また，それらをもとにして部分評価部で部分評価が行われ，結果として，プログラムの一部の機能のみを実現するプログラムが生成される．

スライスを抽出する場合には，ユーザは，まず，スライサ部に，どの変数に関するスライスを抽出するのかといった情報を入力する．その後，ツール内では，そのスライス起点情報とPDGがスライサ部に渡され，そこでスライスが計算される．そのスライス結果は，元のプログラムの部分プログラムとして出力される．

また，部分評価部やスライサ部から出力された部分プログラムもインタープリタ部で実行することができる．この場合にも上述のようなデバッグ作業を行うことができる．このようにして，デバッグの対象を限定することにより，効率良くデバッグ作業を進めることができる．

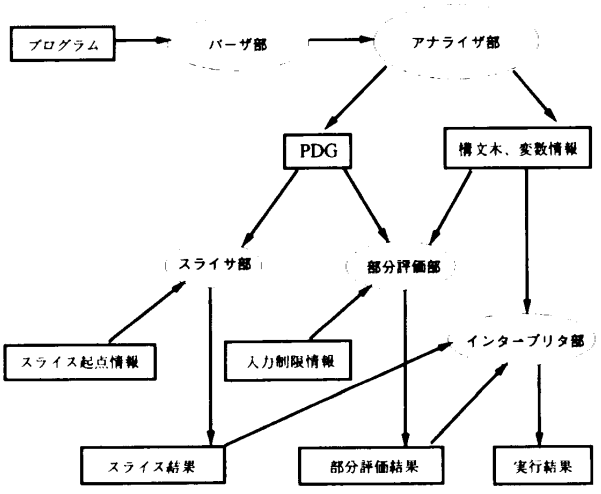

図5 ツールの構成図
Fig. 5 Components of this tool.

4. 実行および評価

4.1 簡単な例

本ツールの実行例として，**図6**に示されるバグを含むプログラムをデバッグする例を考える．このプログラムは，入力された5つの数の合計，最大，最小，平均を計算し，出力するプログラムである．

```
 1  program coverage(input,output);
 2    var   Sum, Max, Min, Mean: integer;
 3          A: array[1..5] of integer;
 4          n: integer;

 5  procedure calc(var sum, max, min, mean: integer);
 6    var   i: integer;
 7    begin
 8      i := 1 + 1;
 9      while i <= n do
10        begin
11          sum := sum + A[i];
12          if A[i] > max then
13            max := A[1]
14          else if A[i] < min then
15            min := A[i];
16          i := i + 1
17        end;
18      mean := sum div n;
19     end;

20  begin
21    n := 5;
22    writeln('Please Input ',n,' values');
23    readln(A[1],A[2],A[3],A[4],A[5]);
24    Sum := A[1];
25    Max := A[1];
26    Min := A[1];
27    Mean := A[1];
28    calc(Sum, Max, Min, Mean);
29    writeln('SUM      : ', Sum);
30    writeln('MAX      : ', Max);
31    writeln('MIN      : ', Min);
32    writeln('MEAN     : ', Mean)
33  end.
```

図6 バグを含んだプログラム
Fig. 6 Example of program including a bug.

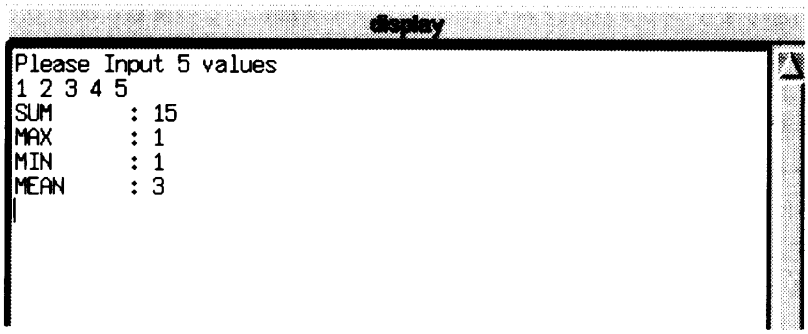

図7 プログラムの実行結果
Fig. 7 Output of the program.

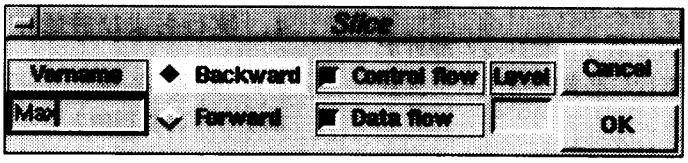

図8 スライス選択画面
Fig. 8 Select window.

(1) プログラムを解析し，インタープリタ部で実行させる．この実行結果が図7である．これを見ると，明らかに，最大値を示すMAXの値が誤っている．

(2) この値を出力した変数（図6における30行目の変数Max）に関するスライス（backward slice）を抽出する．このとき，図8のようなウィンドウが現れる．この画面で，スライスを抽出したい変数名や，CDおよびDDのどちらの依存関係についてPDGを探索するか，PDGを探索するパスの長さなどの情報を入力する．ここでは，変数Maxに関して，CD，DDの両方の依存関係についてPDGをできる限り探索してbackward sliceを求めることにする．こうして得られるスライス結果は図9中の網掛け部分として参照される．ここで，スライスの起点となった変数Maxには直接関係のない文（24，26，27行目の文）がスライスに含まれているが，これらは，28行目の手続き呼び出し文で参照している変数の値に影響を与える文であり，この手続き呼出文がスライスに含まれるため，これらの文もスライスに含まれる．

(3) 30行目のMaxに直接定義を行っている文は，13行目の文である．これはスライスを計算する際に，レベルの設定を1などの適当な値に設定することにより得られる．

(4) この文にブレークポイントを設定し，スライス部分のみを実行させる．すると，この文自体は何度も実行されているにもかかわらず，Maxの値が変化していないことが分かる．これは，この文で参照される変数がA[1]となっているためで，それをA[i]に修正することにより正しい結果が得られる．

表1 実行速度
Table 1 Analysis speed.

program	行数	手続き数	解析時間（秒）
wc	48	0	0.09
calendar	84	4	0.22
201	89	4	0.63
arraysum	93	13	0.07
pai	230	13	3.95
hanoi	28	2	0.1
mutual	95	4	0.91

4.2 実行時間

本ツールを用いてプログラムを解析したときの解析時間を表1に示す．この表には，実行したプログラム，その中に含まれるコードの行数および手続き数，および，そのプログラムをSun Sparc Station ELC上で実行させた場合の解析時間（秒）が示されている．この解析時間とは，ツールのパーザ部およびアナライザ部における実行時間であり，スライスの抽出，部分評価の実行時間は含まれていない．

スライスの抽出は，2.2節で述べたように，依存関係解析の結果生成されたPDGを探索することによって行われるが，その時間は比較的短い．たとえば，表1中のプログラムpaiは，解析に3.95秒かかっているにもかかわらず，スライスの抽出にかかった時間は0.02

```
1  program coverage(output);
2    var   Sum, Max, Min, Mean: integer;
3          A: array[1..5] of integer;
4          n: integer;

5  procedure calc(var sum, max, min, mean: integer);
6    var   i: integer;
7    begin
8      i := 1 + 1;
9      while i <= n do
10       begin
11         sum := sum + A[i];
12         if A[i] > max then
13           max := A[i]
14         else if A[i] < min then
15           min := A[i];
16         i := i + 1
17       end;
18     mean := sum div n;
19   end;

20 begin
21   n := 5;
22   writeln('Please Input ',n,' values');
23   readln(A[1],A[2],A[3],A[4],A[5]);
24   Sum := A[1];
25   Max := A[1];
26   Min := A[1];
27   Mean := A[1];
28   calc(Sum, Max, Min, Mean);
29   writeln('SUM      : ', Sum);
30   writeln('MAX      : ', Max);
31   writeln('MIN      : ', Min);
32   writeln('MEAN     : ', Mean)
33 end.
```

図9 スライス結果（網掛けの部分がスライス）
Fig. 9 Result of slicing (slices are shaded).

秒であった．したがって，スライス計算の時間については，ここでは述べない．また，入力されたプログラムを実行する時間は，入力データに依存するため，これについてもここでは述べない．

また，表中のプログラム wc から語数のみを計算する部分を取り出すような部分評価にかかった時間は，4.69 秒であった．これは部分評価の手続きではプログラムを 2 度解析したり，各文の解釈ごとに全大域変数の組からなる状態をチェックしたりするオーバーヘッドによるものである．

4.3 実行結果

スライシングおよび部分評価を行うと，プログラムの参照部分を小さくできる．その割合は，入力プログラムおよび変数の指定方法などに依存するが，たとえば，図 1 のプログラムは，スライシングの結果，38 行のプログラムが 28 行になった．また，図 3 および図 4 のプログラムは，部分評価の結果，46 行のプログラムが，26 行になった．

4.4 考察

デバッグ時に，プログラムのうちバグに関係のある部分のみを抽出してその部分のみをデバッグの対象とすることにより，開発者のプログラムの参照範囲を小さくするという当初の目的は，本ツールを用いることにより実現できる．また，その依存関係情報は，1 度解析すれば内部に保存できるため，同じプログラムに対しては，解析しなおすことなく別のスライス計算などの作業を行うことができる．

ただし，一般に元のプログラムが大きいときには，スライスも大きくなる[1]．本ツールでは，スライス全体を参照するだけでなく，PDG を探索するレベルを任意に設定することができるので，注目している文に直接影響する文のみを抽出したり，データ依存もしくは制御依存の一方だけを用いた PDG の探索もできるため，参照部分をさらに小さくすることができる．今後，プログラム全体ではなく，手続き単位でスライスを計算したり，実行したりできる機構があればさらに便利になると考えられる．

また，部分評価を行うと，式中の変数が定数に置き換えられる場合が起こるが，その場合でも式の意味は変わらない．ただし，部分評価により削除される部分

にバグが含まれていた場合，当然そのバグは部分評価結果には含まれない．しかし，本来，部分評価は入力変数もしくはそれに相当する変数の値を固定することにより，プログラムの機能を限定することなので，部分評価によって削除される部分にバグが存在していたとしても，それはその時点で注目している部分ではない．したがって，その時点でのバグの原因究明には影響を与えない．

5. まとめ

プログラムから依存関係に基づいてスライシングや部分評価を行い，バグに関係のある部分のみを抽出するデバッグ支援ツールを作成した．本ツールを用いることにより，プログラム全体を対象としていた従来のデバッガを用いるよりも効率の良いデバッグを行うことができると期待される．また，本ツールでは，プログラムの依存関係解析だけでなく，部分評価に必要なそのほかの情報を得るための解析も行い，保存している．それらは必要に応じて自由に利用できるので，デバッグ以外にも，それらの情報を利用するような作業（たとえば，保守[7]など）にも役立つと考えられる．

今後の課題としては，ポインタ変数や構造体も扱えるような依存解析アルゴリズム[2),10),13)]を適用し，対象言語を拡張することなどがあげられる．

謝辞 本研究を行うにあたり，多大なご指導をいただいた，奈良先端科学技術大学院大学情報科学研究科の鳥居宏次教授に深謝いたします．

参考文献

1) Agrawal, H., Demillo, R.A. and Spafford, E.H.: Debugging with Dynamic Sllicing and Backtracking, *Software-Practice and Experience*, Vol.23, No.6, pp.589–616 (1993).
2) Choi, J.D., Burke, M. and Carini, P.: Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects, *Conference Record of 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, pp.232–245 (1993).
3) Coen-Porisini, A., Paoli, F.D., Ghezzi, C. and Mandrioli, D.: Software Specialization Via Symbolic Execution, *IEEE Trans. Softw. Eng.*, Vol.17, No.9, pp.884–899 (1991).
4) Fritzson, P., Gyimothy, T., Kamkar, M. and Shahmehri, N.: Generalized Aigirithmic Debugging and Testing, *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Vol.26, No.6, pp.317–326 (1991).
5) Fritzson, P., Shahmehri, N. and Kamkar, M.: Generalized Algorithmic Debugging and Testing, *ACM Letters on Programming Languages and Systems*, Vol.1, No.4, pp.303–322 (1992).
6) 二村良彦ほか：プログラム変換，共立出版，chapter 4, pp.63–79 (1987).
7) Gallagher, K.B. and Lyle, J.R.: Using Program Slicing in Software Maintenance, *IEEE Trans. Softw. Eng.*, Vol.17, No.8, pp.751–761 (1991).
8) Naoi, K. and Takahashi, N.: Detection of Infeasible Paths with a Path Dependence Flow Graph, *Transactions of the Institute of Electronics, Information and Communication Engineers*, Vol.J76D-I, No.8, pp.429–439 (1993).
9) Lyle, J.R. and Weiser, M.: Automatic Program Bug Location by Program Slicing, *Proc. 2nd International Conference on Computers and Applications*, pp.877–883 (1987).
10) Pande, H.D., Landi, W.A. and Ryder, B.G.: Interprocedural Def-Use Associations for C Systems with Single Level Pointers, *IEEE Trans. Softw. Eng.*, Vol.20, No.5, pp.385–402 (1994).
11) Ramalingam, G.: The Undecidability of Aliasing, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.5, pp.1467–1471 (1994).
12) 佐藤慎一，飯田 元，井上克郎，鳥居宏次：プログラムスライスの抽出・実行機能を組み込んだデバッグ支援システムの試作，第49回情報処理学会全国大会論文集 (5), pp.217–218 (1994-9).
13) 佐藤慎一，植田良一，井上克郎：再帰やポインタを含むプログラムの効率的な依存関係解析法の提案，信学技報，Vol.95, No.475, pp.9–16 (1996).
14) Shimomura, T.: Bug Localization Based on Error-Cause-Chasing Methods, *Trans. IPS Japan*, Vol.34, No.3, pp.489–500 (1993).
15) 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).
16) 高谷暢之: 出力の制限情報を利用したプログラム簡素化手法の提案，修士論文，大阪大学基礎工学部情報工学科 (1994).
17) 植田良一，練 林，井上克郎，鳥居宏次: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).

(平成7年8月2日受付)
(平成8年2月7日採録)

佐藤 慎一（学生会員）

昭和46年生．平成6年大阪大学基礎工学部情報工学科卒業．現在同大学大学院基礎工学研究科物理系専攻情報工学分野博士前期課程在学中．プログラムの依存解析の研究に従事．

飯田 元（正会員）

昭和63年大阪大学基礎工学部情報工学科卒業．平成2年同大学大学院博士前期課程修了．同年同後期課程入学．平成3年大阪大学基礎工学部情報工学科助手．平成7年奈良先端科学技術大学院大学情報科学センター助教授，現在に至る．平成6年大阪大学，工学博士．ソフトウェア開発プロセスおよび開発支援環境，協調作業支援技術などの研究に従事．IEEE，日本ソフトウェア科学会各会員．

井上 克郎（正会員）

昭和54年大阪大学基礎工学部情報工学科卒業．昭和59年同大大学院博士課程修了．同年同大基礎工学部情報工学科助手．同年8月ハワイ大学マノア校芸術・科学学部助教授．昭和61年大阪大学基礎工学部情報工学科助手．平成元年同学科講師．平成3年11月同学科助教授．平成7年12月同学科教授，現在に至る．工学博士．ソフトウェア工学の研究に従事．ACM，IEEE，電子情報通信学会各会員．