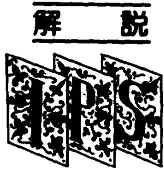
解　説



Program Slicing 技術とテスト，デバッグ，保守への応用†

下　村　隆　夫††

1. はじめに

Program Slicing はプログラム内の文の間の依存関係を明らかにする技術であり，Maryland 大学の Mark Weiser 博士によって1982年に初めて考案されたものである．最初はプログラムのデバッグを支援するために考えられていたが，今日では，テスト，デバッグ，保守へと広範囲に応用されている有用な技術である．本稿では，2. で Program Slicing 技術とは何かを説明し，3. でデバッグへの応用について，4. でテストへの応用の一つとして，テストデータの自動生成について，最後に，5. では保守への応用として，プログラムの改造を支援する二つの方式について述べる．

2. Program Slicing 技術

Program Slicing はプログラム内のある文の実行に影響を与える全ての文を抽出する技術であり，抽出された文の集合を Slice と呼ぶ．Slice には，プログラムを静的に解析して得られる Static Slice と，動的に解析して得られる Dynamic Slice の二つがある．ここでは，説明を簡単にするため，プログラムは，代入文，if 文，while ループ文の3種類の文のみをもつとする．

2.1 Static Slice

Static Slice は，ある文の実行に影響を与える可能性のある文の集合である[1)~5)]．

まず，文の間に次の二つの依存関係を導入する．

- Data Dependence

文 $s1$ から文 $s2$ への Data Dependence 関係があるとは，文 $s1$ における，ある変数 v の定義が，v を使用している文 $s2$ に到達する場合．すなわち，1) 文 $s1$ において変数 v を定義している（文 $s1$ で変数 v に値を設定している），かつ，2) 文 $s2$ において v を使用している（文 $s2$ で変数 v の値を参照している），かつ，3) v を再定義しない，文 $s1$ から文 $s2$ への実行可能なパスが存在する場合である．

- Control Dependence

文 $s1$ から文 $s2$ への Control Dependence 関係があるとは，文 $s1$ は if 文か while ループ文であり，文 $s2$ の実行の有無が文 $s1$ の実行結果に直接依存する場合．

文 s における変数 v に関する Static Slice とは，Data Dependence 関係，あるいは Control Dependence 関係を辿って，文 s の変数 v に到達する全ての文の集合である（図-1）．文 s に関する Static Slice とは，上記の二つの関係を辿って，文 s 内で使用しているいずれかの変数に到達する全ての文の集合である．文の集合 $S=\cup_i s_i$ に関する Static Slice は，$\cup_i$ (s_i の Static Slice) で定義される．Static Slice の例を図-2 に示す．文 s の Static Slice は，文 s の実行結果を保存する完全なプログラムを構成する（もちろん，必要なデータ宣言などを含める）という特徴をもつ．

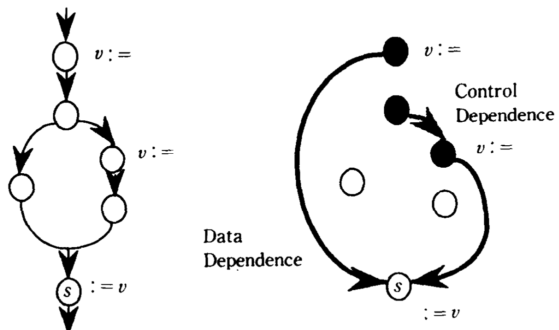


図-1　文 s における変数 v に関する Static Slice の求め方

† Program Slicing Technique and Its Application to Testing, Debugging and Maintenance by Takao SHIMOMURA (NTT Software Laboratories).
†† NTT ソフトウェア研究所

2.2 Dynamic Slice

Dynamic Slice は，ある入力に基づいてプログラムを実行したとき，ある文の実行に実際に影響を与えた，実行した文の集合である[6)~8)]．実行された文を繰り返しも許して実行された順に並べたものを実行系列と呼び，その要素を X^p（p 回目の文の実行時に，実行された文 X）で表す．Dynamic Slice は，この実行系列内から，ある文の実行に実際に影響を与えた実行された文を抽出したものである．Static Slice のときと同様に，まず，実行された文の間に次の二つの依存関係を導入する．

- Data Influence

文 X^p から文 Y^q への Data Influence 関係があるとは，文 Y^q の実行で使用したある変数を最後に定義した文が文 X^p である場合．
すなわち，1) $p<q$，2) 文 X^p の実行において変数 v を定義した，3) 文 Y^q の実行において v を使用した，かつ，4) $p<k<q$ となる任意の k に対して，文 Z^k は v を定義しなかった場合である．

- Control Influence

文 X^p から文 Y^q への Control Influence 関係があるとは，文 X^p は if 文か while ループ文であり，文 Y^q の実行の有無を直接決定した文である場合．

文 X^p における変数 v に関する Dynamic Slice とは，Data Influence 関係，あるいは Control Influence 関係を辿って，文 X^p の変数 v に到達する実行系列内の文の集合である．図-7 に示すプログラムに入力 $n=2$, $a=(2,4)$ を与えて実行したときの，出力文における変数 sum に関する Dynamic Slice の求め方を図-3 に示す．

Dynamic Slice は，ある特定の実行における文の間の依存関係を分析したい場合に有効である．Static Slice と Dynamic Slice との違いを図-4 に示す．文 $s2$ における変数 v に関する Static Slice と Dynamic Slice について考える．文 $s1$ における変数 v の定義は v を使用している文 $s2$ に到達するので，文 $s1$ は Static Slice に含まれる．しかし，文 $s1$ は文 $s2$ で使用した変数 v を最後に定義した文ではないため，Dynamic Slice には含まれない．

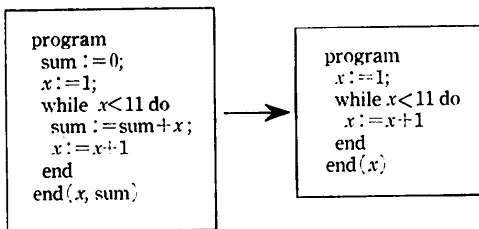


図-2 end 文における変数 x に関する Static Slice

3. デバッグへの応用

プログラムのデバッグにおいては，現状では各種のシンボリック・デバッガ[9)~12)]が使われている．プログラムの実行状況を図形を用いて分かりやすく表示できるもの[13)~15)]や，特定のイベントの発生を容易に検出できるもの[16)~18)]もある．しかし，これらのシンボリック・デバッガでは，バグの存在箇所を究明する作業はプログラマの知識と経験に基づいている．一方，Program Slicing

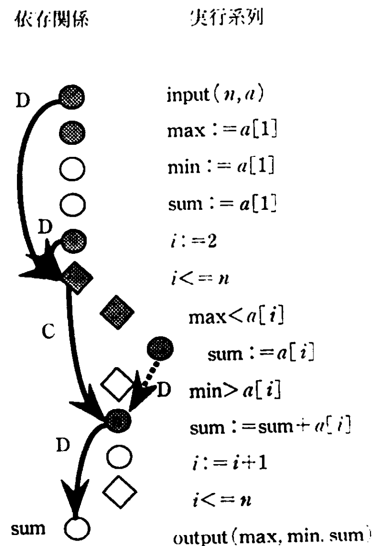


図-3 出力文における変数 sum に関する Dynamic Slice の求め方

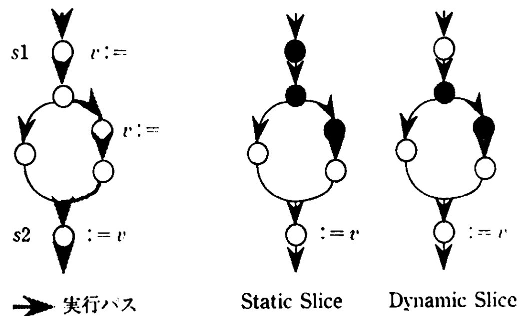


図-4 Static Slice と Dynamic Slice の比較

ではプログラム内のある文の実行に影響を与えた全ての文を抽出することができる．したがって，プログラムを実行した結果，ある変数の値が誤っていることが分かった場合には，その変数に値を設定した文に関する Slice を求めることにより，誤った値を生成した部分を限定することができる．Static Slice や Dynamic Slice を利用したデバッグシステムとしては，Focus[4),5)], Spyder[19),20)], STAD[21)], PELAS[22),23)], CHASE[24)] などがある．ここでは，あらかじめ決められた手順に基づいてバグを見つけるデバッグシステム PELAS について紹介する．

3.1 アルゴリズミック・デバッギング (PELAS)

アルゴリズミック・デバッギングは Ehud Shapiro が初めて発表した技術であり，論理型／関数型言語に対して適用できるものであったが[25)]，PELAS では Dynamic Slice を利用することにより，PASCAL, C などの手続き型言語に対して，あらかじめ決められた手順に基づいてバグを見つける技術を実現している．

（1） Potential Influence

PELAS では，ある変数の値が誤っている場合，実行系列を逆向きに順に辿ってバグのある文を究明するが，このために，Data Influence (DI) と Control Influence (CI) のほかにもう一つの依存関係 Potential Influence (PI) を導入している．

- Potential Influence

文 X^p から文 Y^q への Potential Influence 関係があるとは，文 X^p は if 文か while ループ文であり，文 X^p の実行結果が変われば，文 Y^q で使用したある変数の値を変える可能性がある場合．

すなわち，1) 文 X^p は if 文か while ループ文である，2) $p<q$, 3) 文 Y^q の実行において変数 v を使用した，4) $p<k<q$ となる任意の k に対して，文 Z^k は v を定義しなかった，かつ，5) 文 X^p の実行結果が変わった場合の文 X から文 Y への実行可能なパスで，変数 v を定義するものが存在する場合である (**図-5**)．

（2） デバッグのガイド手順

ある変数の値が誤っている場合には，まず，その Potential Influence と Data Influence の和集合 S を求める．次に，S 内の文 X^p を一つ取り出し，X^p における制御フローが正しいかどうかを調べる．制御フローが誤っていれば，その Control Influence を調べていけばよい．制御フローが正しく，X^p の実行結果が誤りである場合には，X^p で使用している変数の値 w で誤っているものがないかどうかを調べる．誤っているものがあれば，その Potential Influence と Data Influence を再び求めていく．誤っている値がなければ，文 X^p をバグとして検出する．このアルゴリズムでは，誤った値に対する Potential Influence や Data Influence が存在しない場合，および，存在しても，それらの実行結果が正しい場合には，バグを発見できず，この手続きは失敗する (**図-6**)．

（3） デバッグ例

図-7 に示すプログラムを実行したときのデバッグ例を**図-8** に示す．変数 a は配列で 2,4 という二つの値が入力され，変数 n には値 2 が入力されたものとする．図-8 の右端の欄は，各文の実行で定義，あるいは使用された変数の値を示している．プログラムを実行した結果，sum の値が 8 になっている（正しい値は 6 である）．そこで，まず，最後の出力文における変数 sum に関する Potential Influence (PI) と Data Influence (DI) を求める．PI は，while ループ文の条件式 ($i<=n$) である．ここでの制御フローは正しく，実行結果も誤っていない．DI は，代入文 (sum :=sum+$a[i]$) である．この文の実行結果 (sum=8) は誤っているが，それは使用している変数 sum の値 4 が誤っているためである．そこで，次に，この sum の値に関する PI と DI を求める．DI は，代入文 (sum :=$a[i]$) である．この文の実行結果

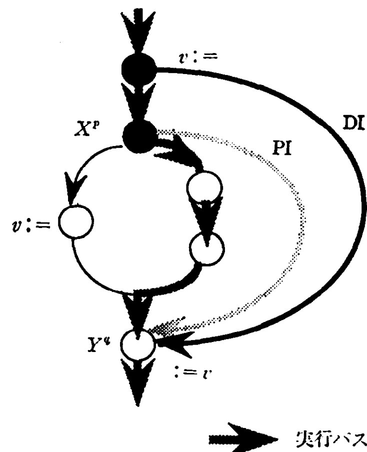


図-5 Potential Influence

(sum=4) は誤っているが，使用している値 ($a[2]=4$; $i=2$) で誤っているものがない．そこで，この代入文がバグとして検出される（正しい文は max :=$a[i]$; である）．

4. テストへの応用

Program Slicing 技術のテストへの応用として，テストデータの自動生成を効率的に行うために Dynamic Slice を利用する例を紹介する．

4.1 テストデータの自動生成 (TestGen)

記号実行によってテストデータを自動生成する手法がいくつか報告されているが[26)~28)]，TestGen[29),30)] では，ある初期入力に基づいてプログラムを実際に実行し，あらかじめ選択されていたパスを実行しないような分岐箇所に達した時点で，パスに沿って実行するように入力変数の値を変更していくという手続きを繰り返す．

（1） テストデータの生成手順

まず，あるテスト基準に基づいて，実行すべきパスを選択する．パスを選択するためのテスト基準としては，ブランチ・テスト[31),32)]，パス・テスト[33),34)]，データフロー・テスト[35),36)]などがある．パスが選択されたら，そのパスに沿ってプログラムを実行させる入力データを求める．

（2） テストデータ生成アルゴリズム

入力変数 x にある初期値を与えてプログラムを実行する．二つの分岐箇所 B_1: if i<high then... と B_2: if max<$A[i]$ then... があり，どちらも条件式が True となって実行されたとする．しかし，選択されたパスに従うと，B_2 では False とならなければならなかったとする．パスどおりに実行するための条件は $\max \geq A[i]$ である．すなわち，$F(x)=A[i]-\max \leq 0$ となる．そこで，入力変数 x の値を少しずつ変えて，それまでに通過した分岐箇所の実行結果を保存し，かつ，$F(x)$ の値を最小にするような値 x を求める．$F(x) \leq 0$ となる値 x が求まれば，その値に基づいて，再びプログラムを実行し，選択されたパスどおりに実行されない次の分岐箇所を見つけ，同様の処理を継続する．

（3） テストデータの生成例

サンプルプログラムと選択されたパスを図-9に示す．パス上の○印は，与えられた各入力に基づいてプログラムを実行した結果，選択されているパスどおりに実行されなかった，最初の分岐箇

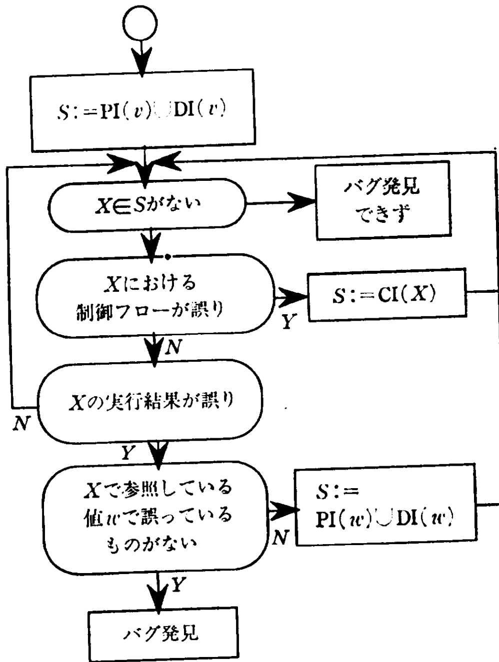


図-6 デバッグ手順

```
input(n, a);
max := a[1];
min := a[1];
sum := a[1];
i := 2;
while i <= n do begin
  if max<a[i] then
    sum := a[i];
  if min>a[i] then
    min := a[i];
  sum := sum+a[i];
  i := i+1;
end;
output(max, min, sum);
```

図-7 デバッグ例（1）

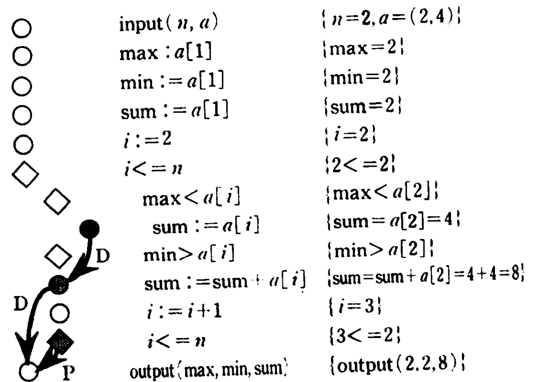


図-8 デバッグ例（2）

所を表す．まず，**図-10** に示す初期入力値に基づいてプログラムを実行する．すると，分岐箇所 if max<$A[i]$ then... で選択されたパスどおりに実行されない．そこで，$F_1(x)=A[i]-\max\leq 0$ の値を最小にするように，入力変数 $A[1]$，$A[2]$，…，$A[100]$，high，step，low の値を順に変えていく．$A[1]$ から $A[38]$ までの入力変数の値を変えても $F_1(x)$ の値は変化しない．$A[39]=139$ とすると，初めて，$F_1(x)\leq 0$ となり，パスどおりに分岐することができる．同様に実行を続けると，最終的に，$A[39]=51$，$A[63]=-37$，high$=67$ とすると，選択されたパスに沿って実行できることがわかる．

（4） Dynamic Slice の利用

選択されたパスどおりに実行されなかった最初の分岐箇所では，$A[1]$，$A[2]$，$A[3]$，… の値を順に変えて，$F_1(x)$ の値が小さくなるような値を求めた．しかし，$A[1]$ から $A[38]$ までの入力変数の値を変えても $F_1(x)$ の値は変化しなかった．分岐箇所 if max<$A[i]$ then… に関する Data Influence を求めると，分岐条件式の中で使用している変数 max や $A[i]$ の値に実際に影響を与えた入力変数を求めることができる．この場合には，$A[39]$，$A[51]$，step，low が影響を与えている．そこで，最初からこれらの入力変数の値を変えることにより，$F_1(x)\leq 0$ とする入力変数の値を効率良く求めることができる（**図-11**）．

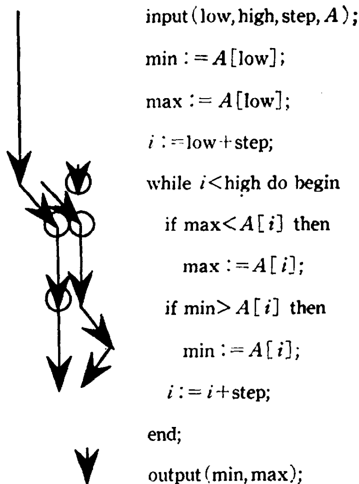


図-9 テストデータ生成例（1）

入力変数	$A[1]$	$A[2]$	...	$A[100]$	high	step	low
初期値	1	2		100	93	12	39

$F_1(x)=A[i]-\max\leq 0$ ......$A[39]=139$
$F_2(x)=\min-A[i]\leq 0$ ......$A[39]=51$
$F_3(x)=A[i]-\max\leq 0$ ....$A[63]=-37$
$F_4(x)=\text{high}-i\leq 0$ ....high$=67$

図-10 テストデータ生成例（2）

5. 保守への応用

Program Slicing 技術の保守への応用として，影響を波及させない部分的改造をガイドするエディタ，および二つの独立した改造版を自動的に統合するシステムを紹介する．

5.1 影響を波及させない部分的改造をガイドするエディタ (Surgeon)

あるプログラムが A, B, C の三つの機能をもつとき，B, C の機能は変えないで，A の機能を A′ に改造したい場合がある．Surgeon[37)~39)] では，B, C の機能に影響を波及させないようにAを改造する方法を提供する．したがって，改造後のテストは改造対象 A′ についてのみ行えばよく，改造対象でない機能 B, C の再テストが不要になるという利点がある．

（1） 改造の手順

まず，改造対象機能に関する Static Slice S_M，および，改造対象外機能に関する Static Slice S_U を取り出す．S_M-S_U に含まれる文を独立な文，S_U を Complement と呼ぶ．Complement を編集禁止にし，独立な文に対して Complement に影響を与えないような編集のみを許可する．Complement に影響を与えない編集には，1)独立な文の削除，2)新しい変数の導入，3)独立な変数へ値を設定する代入文の追加，4)制御の流れを変えない if 文の追加などがある．

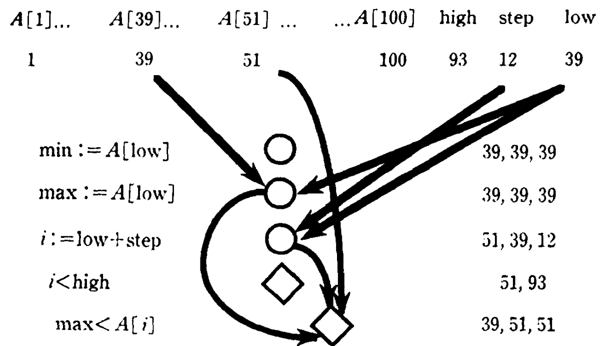


図-11 テストデータ生成例（3）

（2） 改造の例

図-12 に示すプログラムでは，行数 (nl)，語数 (nw)，文字数 (nc) をカウントしている．このプログラムでは，空白で区切られた文字列を語として認識している．行数，文字数のカウント方法は変えないで，アルファベットのみを語と認識するように，このプログラムを改造する．Surgeon エディタのウィンドウには，改造対象機能 (nw) に関する Static Slice S_M が表示され，その中の Complement 部分は反転表示され編集禁止となる．改造した結果を**図-13** に示す．○印は，追加された文を表す．

5.2 二つの独立した改造版の統合 (Integrate)

マシンやオペレーティング・システムの違いにより，あるプログラムに対していくつかの版を作成しなければならない場合がある．このプログラムにバグがあって修正する場合には，それらの全ての版について同じ修正を施す必要がある．Integrate[40),41)] では，二つの独立した改造版を統合することにより，このような修正を自動化することができる．

（1） 統合の手順

あるプログラム Base と，その改造版 A，B に対して，A，B による変更を保存し，かつ，Base の非改造部分の機能も保存するように，Base，A，B の三つのプログラムを統合する．ただし，ある変数の値の計算が Base，A，B でおのおの異なる場合には，A，B は干渉するといい，この場合には統合は失敗する．統合では，プログラム P の各文をノード，文の間の Control Dependence 関係および Data Dependence 関係をアークとするグラフ G_P (Program Dependence Graph) を用いる．図-15 のプログラム Base の Program Dependence Graph (PDG) を**図-14** に示す．統合は以下の手順で行う．

1) PDG 上で変更部分および非改造部分の和集合 G_M をとる．

Base，A，B の PDG を，おのおの，G_{Base}，G_A，G_B とする．A，B の文で，その Static Slice が対応する Base の文の Static Slice と異なるものを Affected Point と呼ぶ．Base に対する A の Affected Point の集合を $AP_{A,Base}$，Base に対する B の Affected Point の集合を $AP_{B,Base}$ で表す．G_A 上で $AP_{A,Base}$ の Static Slice をとり，これを $G_A/AP_{A,Base}$，G_B 上で $AP_{B,Base}$ の Static Slice をとり，これを $G_B/AP_{B,Base}$ とする．また，Base，A，B において，そ

```
inword=NO;
nl=0; nw=0; nc=0;
c=getchar();
while (c!=EOF) {
 nc=nc+1;
 if (c=='\n')
  nl=nl+1;
 if (c==' '||c=='\n')
  inword=NO;
 else if (inword==NO) {
  inword=YES;
  nw=nw+1;
 }
 c=getchar();
}
output(nl, nw, nc);
```

図-12 改造例（1）

```
○  ch=0;
   nw=0;
   c=getchar();
   while (c!=EOF) {
○   if (isspace(c)
     && isalpha(ch))
     nw=nw+1;
○   ch=c;
    c=getchar();
   }
```

図-13 改造例（2）

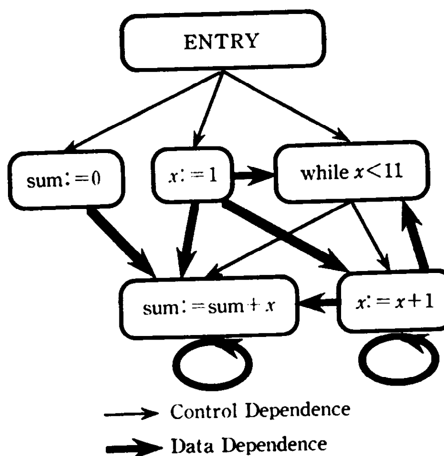


図-14 Program Dependence グラフの例

の Static Slice が同一となる文を Preserved Point と呼び，それらの文の集合を $PP_{Base, A, B}$ で表す．G_{Base} 上で $PP_{Base, A, B}$ の Static Slice をとり，これを $G_{Base}/PP_{Base, A, B}$ とする．G_M は次の式で表される．

$$G_M = (G_A/AP_{A, Base}) \cup (G_B/AP_{B, Base}) \cup (G_{Base}/PP_{Base, A, B})$$

2) A，B の干渉を検査する．

$AP_{A, Base}$ の G_M 上での Static Slice が，G_A 上での Static Slice と一致しない場合には，統合された G_M 上でAによる変更を保存することはできない．したがって，次の条件が成立するときには，A，B は干渉し，統合は失敗する．

$$(G_M/AP_{A, Base} \neq G_A/AP_{A, Base}) \text{ or } (G_M/AP_{B, Base} \neq G_B/AP_{B, Base})$$

3) G_M からプログラム P を再構成する．

G_M 内の各文の間の実行順序（ソースプログラム上での並び順）を決め，プログラムPを作成する．

4) プログラム P の PDG G_P が G_M に一致すれば，統合成功．

（2） 統 合 の 例

サンプルプログラム Base と，その改造版 A，B を**図-15** に示す．○で囲まれた部分が改造された箇所である．改造版Aの Affected Points の G_A 上での Static Slice を**図-16** に示す．統合して得られた G_M を**図-17** に示す．

（3） ノードの並び順

G_M 内の各ノード（文）の並び順を適切に決めないと，ノード間の Data Dependence 関係が変わってしまい，G_M 内の Data Dependence アークを保存できなくなる．たとえば，**図-18** では，文 x:=0；から文 y:=x+w；への Data Dependence 関係，および，文 x:=1；から文 z:=x；への Data Dependence 関係があるが，もし，文 x:=1；を文 y:=x+w；の前に配置してしまうと，文 x:=0；から文 y:=x+w；への Data Dependence 関係は失われてしまう．一般に，ある PDG が feasible であるかどうか（あるプログラムの PDG に一致するかどうか）という問題は，NP 完全であることが知られている[42]．ある変数 x を定義している文 d，および，その文以降でその定義が到達する，x を使用している文の集合を Span (d, x) と呼ぶ．同一変数に対する二つの Span S_1，S_2 は，S_1，S_2 の順か，あるいは，S_2，S_1 の順となり，互いに重なり合うことはないという性質をもつ．同一変数に対する Span 間の順序を適切に決定することにより，実用的に問題のない時間でプログラムの再構成ができることが報告されている[40]．

Base	A	B

program
sum := 0;
x := 1;
while x < 11 do
sum := sum + x;
x := x + 1
end
end(x, sum)
	program
	(prod := 1;)
	sum := 0;
	x := 1;
	while x < 11 do
	(prod := prod* x;)
	sum := sum + x;
	x := x + 1
	end
	end(x, sum, (prod))
		program
		sum := 0;
		x := 1;
		while x < 11 do
		sum := sum + x;
		x := x + 1
		end;
		(mean := sum/10)
		end(x, sum, (mean))

図-15 Base と改造版 A，B

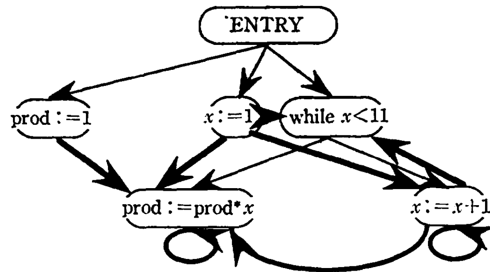


図-16 $G_A/AP_{A, Base}$

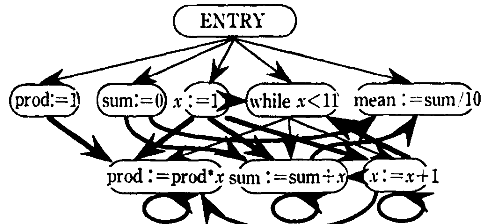


図-17 統合 G_M

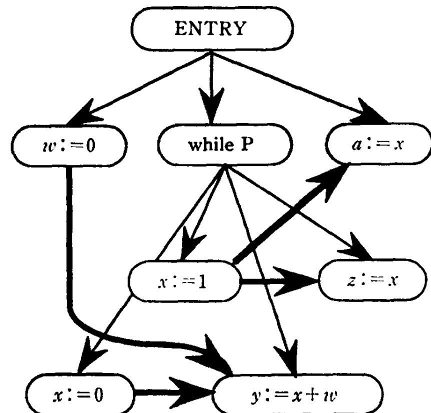


図-18 ノードの並び順

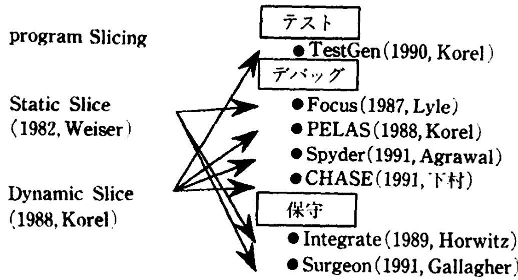


図-19 Program Slicing 技術の応用

6. おわりに

これまでに述べた Static Slice 技術，Dynamic Slice 技術のテスト，デバッグ，保守への応用例をまとめて図-19 に示す．テストデータの自動生成では，プログラムを実際に実行して，選択されたパスどおりに実行されなかった分岐箇所における分岐条件の値に影響を与える入力変数を見つけるために Dynamic Slice 技術が使われた．プログラムのデバッグでは，プログラムの中から誤った値を生成した部分を限定するために，Static Slice, Dynamic Slice の両方の技術が利用できた．プログラムを改造する場合には，ソースプログラムが対象であるため，Static Slice 技術が用いられた．Program Slicing 技術は非常に広範囲な応用をもつ優れた技術であるが，現在はまだ大規模なプログラムを対象とした実用的なシステムはない．今後，さらに応用が広がるとともに，実用的なシステムが開発されることが期待される．

謝辞 本稿をまとめる上で有益な助言をいただきました Maryland 大学の Mark Weiser 博士(現在 Xerox Palo Alto Research Center), James R. Lyle 博士，Loyola 大学の Keith B. Gallagher 博士，ならびに，Wayne State 大学の Bogdan Korel 博士に感謝いたします．また，日頃から励ましをいただいています NTT ソフトウェア研究所所長鶴保征城博士に深謝いたします．

参考文献

1) Weiser, M.: Programmers Use Slices When Debugging, CACM, Vol. 25, No. 7, pp. 446-452 (July 1982).
2) Weiser, M.: Program Slicing, IEEE Transactions on Software Engineering, Vol. SE-10. No. 4, pp. 352-357 (July 1984).
3) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 26-60 (Jan. 1990).
4) Weiser, M. and Lyle, J.: Experiments on Slicing-Based Debugging Aids, Empirical Studies of Programmers, Ablex Publishing Corporation, pp. 187-197 (1986).
5) Lyle, J. and Weiser, M.: Automatic Program Bug Location by Program Slicing, The Second International Conference on Computers and Applications, pp. 877-883 (June 1987).
6) Korel, B. and Laski, J.: Dynamic Program Slicing, Information Processing Letters, Vol. 29, No. 10, pp. 155-163 (Oct. 1988).
7) Agrawal, H. and Horgan, J. R.: Dynamic Program Slicing, ACM SIGPLAN Notices, Vol. 25, No. 6, pp. 246-256 (June 1990).
8) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, J. Systems Software, 13, pp. 187-195 (1990).
9) Bruegge, B. and Hibbard, P.: Generalized Path Expressions: A High-Level Debugging Mechanism, The Journal of Systems and Software, 3, pp. 265-276 (1983).
10) Powell, M. L. and Linton, M. A.: A Database Model of Debugging, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, pp. 67-70 (Mar. 1983).
11) Adams, E. and Muchnick, S. S.: Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations, Software-Practice and Experience, pp. 653-669 (July 1986).
12) Griffin, J. H., Wasserman, H. J. and Mcgavran, L. P.: A Debugger for Parallel Processes, SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 18 (12), pp. 1179-1190 (Dec. 1988).
13) Shimomura, T. and Isoda, S.: Linked-List Visualization for Debugging, IEEE SOFTWARE, pp. 44-51 (May 1991).
14) Moher, T. G.: PROVIDE: A Process Visualization and Debugging Environment, IEEE Transactions on Software Engineering, Vol. 14, No. 6, pp. 849-857 (June 1988).
15) Myers, B. A., Chandhok, R. and Sareen, A.: Automatic Data Visualization for Novice Pascal Programmers, 1988 IEEE Workshop on Visual Languages, pp. 192-198 (Oct. 1988).
16) Lazzerini, B. and Lopriore, L.: Abstraction Mechanisms for Event Control in Program Debugging, IEEE Transactions on Software Engineering, Vol. 15, No. 7, pp. 890-901 (July 1989).
17) Olsson, R. A., Crawford, R. H., Ho, W. W. and Wee, C. E.: Sequential Debugging at a High Level of Abstraction, IEEE Software, pp. 27-36 (May 1991).
18) Olsson, R. A., Crawford, R. H. and Ho, W. W.: A Dataflow Approach to Event-Based Debugg-

ing, SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 21(2), pp. 209-229 (Feb. 1991).

19) Agrawal, H., DeMillo, R. A. and Spafford, E. H.: An Execution-Backtracking Approach to Debugging, IEEE Software, pp. 21-26 (May 1991).

20) Agrawal, H, and Spafford, E. H.: An Execution Backtracking Approach to Program Debugging, Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference, pp. 283-299 (Sep. 1988).

21) Korel, B. and Laski, J.: STAD—A System for Testing and Debugging: User Perspective, Proceedings Second Workshop on Software Testing, Verification, and Analysis, pp. 13-20 (July 1988).

22) Korel, B.: PELAS—Program Error-Locating Assistant System, IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1253-1260 (Sep. 1988).

23) Korel, B. and Laski, J.: Algorithmic Software Fault Localization, Proc. 24th Annual Hawaii International Conference on System Science, pp. 246-252 (1991).

24) Shimomura, T. and Isoda, S.: CHASE: A Bug-Locating Assistant System, COMPSAC '91, pp. 412-417 (Sep. 1991).

25) Shapiro, E. Y.: Algorithmic Program Debugging, The MIT Press (1982).

26) Boyer, R., Elspas, B. and Levitt, K.: SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution, SIGPLAN Notices, Vol. 10, No. 6, pp. 234-245 (June 1975).

27) Clarke, L.: A System to Generate Test Data and Symbolically Execute Programs, IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 215-222 (Sep. 1976).

28) Howden, W.: Symbolic Testing and the DISSECT Symbolic Evaluation System, IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, pp. 266-278 (1977).

29) Korel, B.: Automated Software Test Data Generation, IEEE Transactions on Software Engineering, Vol. 16, No. 8, pp. 870-879 (Aug. 1990).

30) Korel, B., Wedde, H. and Ferguson, R.: Automated Test Data Generation for Distributed Software, COMPSAC '91, pp. 680-685 (Sep. 1991).

31) Stucki, L. G.: Automatic Generation of Self-Metric Software, Proceedings 1973 IEEE Symposium on Computer Software Reliability, pp. 94-100 (1973).

32) Miller, E. F.: Program Testing: Art Meets Theory, IEEE Computer (July 1977).

33) Miller, E. F. et al.: Structurally Based Automatic Program Testing, Proceedings EASCON 74, pp. 134-139 (1974).

34) Howden, W. E.: Reliability of the Path Analysis Testing Strategy, IEEE Transactions on Software Engineering, Vol. SE-2, No. 3 (Sep. 1976).

35) Frankl, P. G. and Weyuker, E. J.: Data Flow Testing in the Presence of Unexecutable Paths, Proc. Workshop on Software Testing (July 1986).

36) Sneed, H. M.: Data Coverage Measurement in Program Testing, Proc. Workshop on Software Testing (July 1986).

37) Lyle, J. R. and Gallagher, K. B.: Using Program Decomposition to Guide Modifications, Proceedings of Conference on Software Maintenance-1988, pp. 265-269 (Oct. 1988).

38) Lyle, J. R. and Gallagher, K. B.: A Program Decomposition Scheme with Applications to Software Modification and Testing, Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, pp. 479-485 (1989).

39) Gallagher, K. B. and Lyle, J. R.: Using Program Slicing in Software Maintenance, IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 751-761 (Aug. 1991).

40) Horwitz, S., Prins, J. and Reps, T.: Integrating Noninterfering Versions of Programs, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, pp. 345-387 (July 1989).

41) Yang, W., Horwitz, S. and Reps, T.: A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations, SIGSOPT SEN, Vol. 15, No. 6, pp. 133-143 (Dec. 1990).

42) Horwitz, S., Prins, J. and Reps, T.: On the Suitability of Dependence Graphs for Representing Programs, Computer Science Dept., Univ. of Wisconsin, Madison (Aug. 1988).

(平成4年1月13日受付)

下村 隆夫(正会員)

昭和24年生．昭和48年京都大学理学部数学科卒業．昭和50年東北大学大学院修士課程修了．NTT ソフトウェア研究所主任研究員．平成4年4月より電気通信大学大学院情報システム学研究科客員助教授．ソフトウェアの設計，テスト，デバッグの自動化に興味をもつ．電子情報通信学会，ACM 各会員．