変数値エラーにおける Critical Slice に基づくバグ究明戦略†

下村 隆夫††

現状のプログラムデバッグ作業では，発見されたエラーを基に，そのエラーを引き起こしている原因についてのいくつかの仮説を立て，それらの仮説を検証することを繰り返しながらバグを究明していく．仮説の作成はプログラマの知識と経験に基づくため，デバッグ作業は試行錯誤で進められる．本論文では，変数値エラーに対して，Critical Slice に基づいた決定性のバグ究明戦略を提案する．Critical Slice はエラーを引き起こす可能性のある文を含む最小の集合であり，この Critical Slice を分割しフローデータの値の正誤を判定することによりバグを究明できることを示す．

1. はじめに

現状のシンボリックデバッガ[1)~7)]を用いたプログラムデバッグ作業では，発見したエラーに関係していると思われる部分にブレークポイントを設定してプログラムを実行し，その部分におけるプログラム状態を調べるという作業を繰り返す．すなわち，発見されたエラーを基に，そのエラーを引き起こしている原因についてのいくつかの仮説を立て，それらの仮説を検証することを繰り返しながらバグを究明していく[8)]．仮説の作成はプログラマの知識と経験に基づくため，デバッグ作業は試行錯誤で進められる．このため，効率よくバグを検出できるかどうかはプログラマの能力に大きく依存している．

一方，システムのガイドに従ってバグを究明できるものもある．Algorithmic Debugging[9)]では，プログラマはシステムから提示された関数の正誤をその入出力値を基に判定する．これを繰り返しながら，次第に誤りを含む部分を限定し，バグを含む関数を検出する．この方式は副作用のある手続き型言語には適用することができない．また，検出できるのはバグを含む関数であり，バグを含む文まで限定することはできない．プログラム依存関係解析方式[10)~17)]では，ある変数の値の誤りを基にして，そのエラーに影響しているかもしれない部分を限定する．しかし，エラーに影響しているかもしれない部分からバグを見つけるための効率的な手法をもたない．

本論文では，変数値エラーに対して，Critical Slice に基づいた，効率のよいバグ究明戦略を提案する．Critical Slice はエラーを引き起こす可能性のある文を含む最小の集合であり，この Critical Slice を分割しフローデータの値の正誤を判定することによりバグを究明できることを示す．

2. プログラムの性質

2.1 プログラムのモデル

デバッグ対象のプログラミング言語としては，Ada, Pascal, C 等の一般の手続き型言語を想定する．ここでは，言語をモデル化し，プログラムは宣言部と関数定義部からなるとする．宣言部は変数や関数の宣言文からなり，関数定義部は宣言部と処理部からなる．そして，処理部は，代入文，分岐文，ループ文，関数呼び出し文，および入出力文からなるとする．関数呼び出しでは，in パラメータと out パラメータがあり，in パラメータでは call 時に値がアーギュメントからパラメータに渡され，out パラメータでは return 時に値がパラメータからアーギュメントに返されるとする．すなわち，procedure $P(X:\text{in}\cdots; Y:\text{out}\cdots)$; とすると，関数呼び出し文 $P(A,B)$; では，次の文を実行するものと考える．

```
call P;
X := A;
.....
B := Y;
return;
```

(例) 文の例

代入文	$X := Y+Z$;
分岐文	if $X>Y$ then…else…end if;
ループ文	while $X>0$ loop…end loop;
関数呼び出し文	$P(A,B)$;
入出力文	get(X); put(X);

分岐文の分岐条件部分 (if $X>Y$) だけを指す場合には分岐条件部，ループ文のループ条件部分 (while X

† Critical-Slice Based Bug-Locating Strategy in Variable Value Errors by TAKAO SHIMOMURA (NTT Software Laboratories).
†† NTT ソフトウェア研究所

>0) だけを指す場合にはループ条件部と呼ぶこととする.

2.2 バグの分類

プログラムが仕様に合わない場合には，ソーステキスト内でその原因となっている部分を見つけ，修正する必要がある．この原因をバグと呼ぶこととする．プログラムのバグは次の4つに分類できる.

- 文記述漏れバグ (SOB)
  文の記述が漏れている場合
- 文記述過多バグ (SXB)
  余分な文の記述がある場合
- 文記述誤りバグ (SEB)
  文の記述に誤りがある場合で，かつ次の名前記述誤りバグを除いたもの
- 名前記述誤りバグ (NEB)
  次のものからなる.
  1) 代入文において，左辺の変数の名前を誤った場合，2) 関数呼び出し文において，呼び出す関数の名前を誤った場合，3) 関数呼び出し文において，out パラメータに対応するアーギュメントの名前を誤った場合，4) 入力文において入力変数の名前を誤った場合.
  これらのバグは，全く別の変数に値を設定するという誤りを引き起こす可能性がある.

(例) 代入文 $X := Y+Z;$ において，$W := Y+Z;$ が正しかった場合には，代入文の名前記述誤りバグである.

バグは宣言部に存在する場合もある．このような場合でも，その宣言に関係する処理部の文のバグとみなし，バグ究明戦略では処理部におけるバグを見つけることとする.

(例) 変数の宣言に漏れがある場合には，その変数に値を設定し，その値を参照する処理部の文の漏れとみなす.

2.3 バグ潜在域

プログラム内にバグがあると，変数に誤った値が設定されたり，制御の流れが変ったりする．ここでは，ある実行時点において変数の値が誤っている場合（変数値エラー）を考える.

(1) バグ潜在域の定義

次の条件を満たす，プログラム内の文の集合Xをバグ潜在域と定義する.

1) 集合X内の文については，いずれの文に文記述誤りバグ，名前記述誤りバグ，文記述過多バグがあっても，発見されたエラーを引き起こす可能性がある.

2) 集合X以外の文については，いずれの文に文記述誤りバグ，文記述過多バグがあっても，発見されたエラーを引き起こすことはない.

〈定理1〉 バグ潜在域

エラーが発見された時，文記述漏れバグがなく，かつバグ潜在域X以外には名前記述誤りバグがない場合には，バグ潜在域 X の中にバグが存在する.

(証明)

バグ潜在域Xの中に文記述誤りバグも名前記述誤りバグも文記述過多バグも存在しないとすると，X以外の文に文記述誤りバグ，あるいは，文記述過多バグがあることになる．しかし，バグ潜在域の定義より，これらのバグは発見されたエラーを引き起こすことにはならないため，矛盾する．(証明終了)

(2) Static Slice

文 s に関する Static Slice は，次の2つの依存関係を辿って，文 s に到達するすべての文の集合である[10),11),13),14)].

1) 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.
2) Control Dependence
   文 $s1$ から文 $s2$ への Control dependence 関係があるとは，文 $s1$ は分岐文かループ文であり，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.

関数 call がある場合には，call 時と return 時に，アーギュメントとパラメータの間で値の受渡しが行われる．この場合には，パラメータやアーギュメントに値を設定する代入文が実行されるとみなして Slicing を行う[12)].

Static Slice では，変数値エラーが発生した場合に，その変数の誤った値に影響を与える可能性のある文の集合を限定できる．しかし，静的に解析するため，実行されなかった文も抽出してしまう.

(3) Dynamic Slice

Dynamic Slice は，ある入力データを与えてプログラムを実行した時，変数が使用された，ある実行時点において，その変数の値に実際に影響を与えた，実行された文の集合である[15)~17)]．実行された文を繰り返しも含めて，順に並べたものを実行系列と呼ぶ．この実行系列の中で，上記の2つの依存関係を辿っていく

ことにより求められる.

Dynamic Slice では，実行系列内の文の間の依存関係に基づいているため，ある分岐文の分岐結果の誤りのために，ある変数に値が設定されず，それが原因で変数値エラーを引き起こしているような場合には，その分岐文，および，その分岐文が依存している文の集合がまるごと検証対象となる実行系列から漏れてしまう（次節参照）.

（4） Critical Slice

本論文では Critical Slice を定義する. Critical Slice は，ある入力データを与えてプログラムを実行した時，変数が使用された，ある実行時点において，その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である.

ある入力データを与えてプログラムを実行し，繰り返しも含めて，n 個の文が実行されている状態を考える．t 番目に文の実行が行われた時点を実行時点 t と呼ぶ．実行時点1から実行時点 n までの実行時点の列を実行系列と呼び，EX で表す.

EX　　$=\langle 1, 2, 3, \cdots, n\rangle$

$S(t)$　　t 番目に実行された文

Def(t)　実行時点 t における文 $S(t)$ の実行で定義された変数の集合

Use(t)　実行時点 t における文 $S(t)$ の実行で使用された変数の集合

（a） $D(t, v)$ の定義

実行時点 t，変数 v に対して，変数 v に最後に値を設定した実行時点 $D(t, v)$ を次のように定義する.

$$D(t, v)=\max\{j\in \mathrm{EX} \mid j<t, v\in \mathrm{Def}(j)\}$$

実行時点 t，変数の集合 V に対して，

$$D(t, V)=\{D(t, v) \mid v\in V\}$$

（b） $C(t)$ の定義

文 A の実行の有無を決定する文の集合 CE(A) を以下のように定義する.

```
if…then
  S1
else
  S2
end if;
while…loop
  S3
end loop;
procedure P(…)is
begin
  S4
end;
```

文 A が $S1$，あるいは $S2$ 部分に記述されている場合には，その分岐条件部を CE(A) に含める．文 A が $S3$ 部分に記述されている場合には，そのループ条件部 CE(A) に含める．文 A が $S4$ 部分に記述されている場合には，その関数の呼び出し文を CE(A) に含める．また，ループ文 L 自身は，$L\in$CE(L) とする.

実行時点 t に対して，実行時点 t の実行の有無を決定する文を実行した時点の集合 $C(t)$ を次のように定義する.

$$\max\{i\in \mathrm{EX} \mid i<t, S(i)\in \mathrm{CE}(S(t))\}\in C(t)$$

$i\in C(t)$ ならば，$\max\{j\in \mathrm{EX} \mid j<i, S(j)\in \mathrm{CE}(S(i))\}\in C(t)$

（c） $O(t, v)$ の定義

実行時点 t，変数 v に対して，実行系列 EX の部分集合 $O(t, v)$ を次のように定義する.

$O(t, v)=\{j \mid j, k\in \mathrm{EX}, j<k\leq t$ となる，ある k が存在して，$S(j)$ から $S(k)$ への実行されなかったパスで，変数 v に値を設定する文を含むものがあり，$k\leq l<t$ となる，任意の l について，$v\notin \mathrm{Def}(l)\}$

$O(t, v)$ は，実行された分岐条件部，あるいはループ条件部で，その分岐結果が変われば，変数 v の値を設定する可能性があり，それにより，実行時点 t における変数 v の値を変えたかもしれない実行時点の集合である.

（例） あるプログラムに入力データを与えて実行した時の実行パスを**図1**のフローグラフ上に示す. $O(F, v)$ を調べてみると，文 B, D, E は条件を満たすので，$O(F, v)$ に含まれる．文 A は，文 A から文 B に至る，実行されなかったパスで変数 v を定義する文を含むものがあるが，文 B から文 F に至る実行パスの途中に変数 v の定義があるため，$O(F, v)$ には含まれない．文 C は，実行されなかったパスに変数 v を定義する文がないため，$O(F, v)$ に含まれない.

実行時点 t，変数の集合 V に対して，

$$O(t, V)=\{O(t, v) \mid v\in V\}$$

（d） Critical Slice CS(t, v) の定義

実行時点 t における，変数 v に関する Critical Slice CS=CS(t, v) を以下のように定義する.

実行時点 $i=t$ の時には，$V(i)=\{v\}$,

実行時点 $i\neq t$ の時には，$V(i)=$Use(i) とする.

1) $D(t, v)\cup O(t, v)\cup C(t)\subset$CS

2) $i\in$CS ならば，$D(i, V(i))\cup O(i, V(i))\subset$CS

3) $j\in D(i, V(i))$ となる $i, j\in$CS が存在する場合，

$k \in C(j)$ かつ $k \notin C(i)$ ならば，$k \in CS$ とする．

（e） 関係 $Rcs(g, h)$ の定義

$g, h \in CS(t, v) \cup \{t\}$ に対して，g と h が関係 $Rcs(g, h)$ にあるとは，以下の場合をいう．

$g \in D(h, V(h)) \cup O(h, V(h)) \cup C(h)$

（例） 一部分が示されている，次のプログラムをある入力データを与えて実行した時，文 23 において変数 v の値が誤っている場合を考える．実行されたパスを**図 2** のフローグラフ上に示す．

```
11  a := i+3;
12  if i>0 then
13   if j<0 then
14    b := k/2;
     end if;
15   c := b-2;
    end if;
16  v := i+k;
17  d := m+3;
18  if a>0 then
19   if k>0 then
20    e := c+k;
21    if e>0 then
22     v := d+i;
      end if;
     end if;
    end if;
23  n := v-2;
```

Critical Slice $CS(23, v)$ を求める．$16 \in D(23, v)$，$21 \in O(23, v)$ であるので，文 16，文 21 は Critical Slice に含まれる．$20 \in D(21, e)$，$15 \in D(20, c)$，$12 \in C(15)$ であるので，文 20，文 15，文 12 も Critical Slice に含まれる．また，$13 \in O(15, b)$ であるので，文 13 も Critical Slice に含まれる．

文 18，文 19 は，$C(20)$ に含まれるが，Critical Slice には含まれない．このため，$11 \in D(18, a)$ である文 11 も Critical Slice には含まれない．PELAS，STAD[18)~20)] では，これらの文も検証対象としているが，これらの文の実行結果は文 23 における変数 v の値に影響しない．Dynamic Slice では，実行系列内の文の間の依存関係に基づいているため，文 13，文 21 が Slice から漏れてしまう．さらに，文 21 が漏れるために，文 20 も Slice から漏れてしまう．したがって，これらの漏れた文に誤りがある場合には，その検

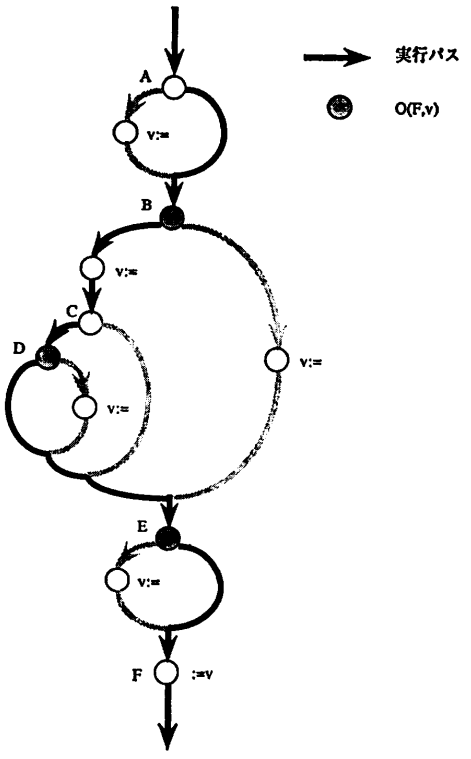


図 1 $O(F, v)$ の例
Fig. 1 Example of $O(F, v)$.

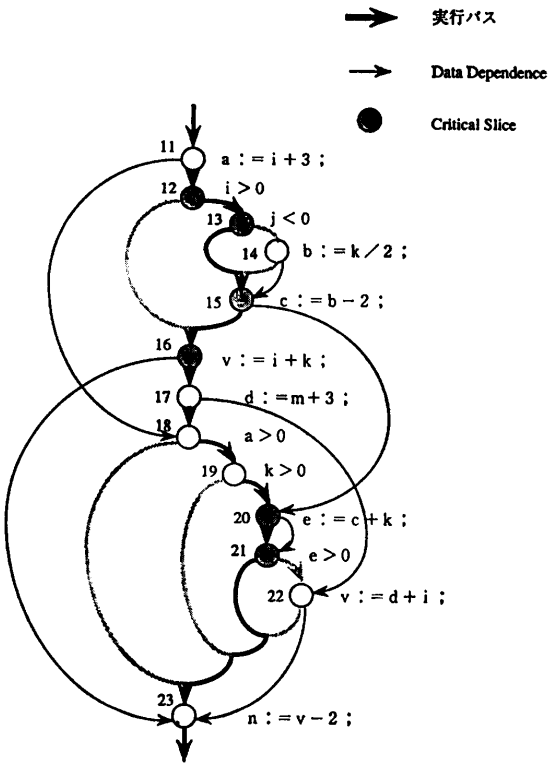


図 2 Critical Slice
Fig. 2 Critical slice.

出が困難になる．Static Slice では，文 11 から文 22 までのすべての文が Static Slice に含まれる．Static Slice, Dynamic Slice, PELAS, Critical Slice の間には，次の関係がある．

Dynamic Slice⊂Critical Slice⊂PELAS
⊂(Static Slice∩EX)

ここで，Static Slice∩EX は，$\{j \in EX \mid S(j) \in$ Static Slice$\}$ を意味する．

(5) 正しい制御フロー

ある実行時点 t における，制御フロー CF(t) を次のように定義する．

$CF(t) = \{t_1 < t_2 < \cdots < t_m \mid 1 \leq k \leq m, t_k \in C(t)\}$

ある実行時点 t における制御フローが正しいとは，CF(t) 内の各実行時点における制御移行が正しい場合をいう．変数値エラーでは，その実行時点における制御フローは正しいが，その変数の値が誤っている場合と，その実行時点における制御フローが誤っている場合がある．

〈定理 2〉 Critical Slice

変数値エラーの場合には，その実行時点 t における，値の誤っている変数 v に関する Critical Slice がバグ潜在域となる．

(証明) 付録 1 に示す．

3. バグ究明戦略

プログラマのバグ究明能力として「ある時点における制御フローが正しい場合には，プログラマはその時点における変数の値の正誤を判断できること」を仮定する．ある時点における制御フローが誤っていると，実行されるべきでない文が実行されているため，その文で定義している変数の値が正しいかどうかを判定することは非常に困難である．

(1) フローデータ

変数値エラーの場合には，定理 2 より，Critical Slice がバグ潜在域となる．したがって，Critical Slice に含まれるどの文にもバグを含む可能性がある．Critical Slice をある実行時点 i の直後で分割する．分割点 i における，フローデータ $F(i)$ を次のように定義する．

$F(i) = \cup V(s)$

$r \leq i < s$ となる，ある $r, s \in$ Critical Slice が存在して，$r \in D(s, V(s))$ の時．

ただし，すべての変数は，プログラムの実行開始時点，あるいは，関数の実行開始時点において未定義値が設定されているものとする．

〈定理 3〉 フローデータ

変数値エラーの場合に，Critical Slice をある実行時点 i の直後で分割する．そして，実行時点 i の直後における制御フローが正しい場合を考える．分割点 i におけるフローデータを $F(i)$ とする．

(a) $F(i)$ 内のある変数の値が誤っていれば，i 以前にバグがある．

(b) $F(i)$ 内のすべての変数の値が正しければ，i より後にバグがある．

(証明) 付録 2 に示す．

したがって，まず，Critical Slice を求め，制御フローの正しいある実行時点でそれを分割し，フローデータの値の正誤を判定する．定理 3 より，これを繰り返すことによりバグを究明することができる．Critical Slice を分割する場合には，文単位では文脈の把握が困難であるため，ブロック単位で行う．ブロックは，ソースプログラム内の，順に実行され，あるまとまった機能を実現しているソース文の集合であるが，ブロックの厳密な定義やブロックへの分解方法については紙面の都合で詳細に述べることができないため，以下では例で示すにとどめる．

(2) ブロック実行系列の検証

Critical Slice 内の実行系列をブロック単位に分割する．ブロック単位に分割した実行系列をブロック実行系列と呼ぶ．ブロック実行系列の実行順序が正しいかどうかを，入力における入力値に基づいて確かめる．分岐条件部，ループ条件部，あるいは call 文だけからなるブロックでは，その制御移行結果も同時に確かめる．

実行系列の検証結果は以下の場合に分けられる．

(a) ブロックの記述が誤っている，ブロックの記述が漏れている，あるいは，ブロックの記述が過多である場合．

バグが検出された．ソースファイル内の対応するブロックを表示して，修正を支援すればよい．

(b) 途中の制御移行結果が誤っている場合．

(b-1) 制御移行した分岐条件部，ループ条件部，あるいは，call 文が Critical Slice に含まれる時．

さらに，次の 2 つの場合に分けられる．

(b-1.1) 分岐条件部，ループ条件部の文記述誤りバグ，あるいは call 文の名前記述誤りバグである．

(b-1.2) 参照している変数の値が誤っている．

この場合には，誤っている値を基にして，次の「フローデータの検証」を行う．

(b-2) 制御移行した分岐条件部，ループ条件部，あるいは，call 文が Critical Slice に含まれない時．

この分岐文，ループ文，あるいは，関数呼び出し文ブロックの直前のフローデータが正しく，かつ，直後における誤ったフローデータを w とすると，このブロック内に変数 w に値を設定する処理が漏れている．すなわち，文記述漏れバグ，あるいは，名前記述誤りバグが存在する．

制御移行した分岐条件部，ループ条件部，あるいは，call 文は Critical Slice に含まれないため，制御移行が誤っていても，現在発生している変数値エラーの原因とはならない．制御移行が誤っている原因を追跡することは，別のエラーの原因を追跡することになる．ここで重要なのは，このブロック内に文記述漏れバグ，あるいは，名前記述誤りバグが存在することを見逃さないことである．

(c) ブロック実行系列の実行順序が誤っていない場合．

誤っている変数値を基にして，次の「フローデータの検証」を行う．

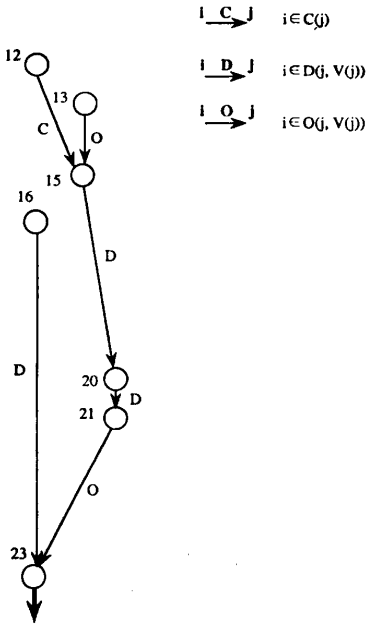


図 3 Critical Slice グラフ
Fig. 3 Critical slice graph.

(3) フローデータの検証

Critical Slice CS(t, v) は，次の定義によりグラフ $G=(N, A)$ となる．

N は，ノードの集合で，$N=\mathrm{CS}(t, v)\cup\{t\}$．$A$ はアークの集合で，$R_{cs}(p, q)$ の時，$(p, q)\in A$ とする．

図2の Critical Slice のグラフを図 3 に示す．

ブロック実行系列に含まれるノードの中，制御移行結果が正しいと判定された分岐条件部ノード，ループ条件部ノード，call 文ノード，および，それらのみに到達するノードは，エラーに影響を与えないため，検証対象から除外する（図 4）．この縮退された Critical Slice (Critical フロー) を分割し，フローデータの値の正誤を判定する．これを繰り返すと，最後には，ブロック直前のフローデータは正しく，かつ，ブロック直後のフローデータで誤ったものがある，1つのブロック (Critical ブロック) が特定される．

(a) Critical ブロック内の Critical フローが空である場合には，誤ったフローデータ w に値を設定する処理が Critical ブロック内に漏れている．すなわち，Critical ブロック内に文記述漏れバグ，あるいは，名前記述誤りバグがある．

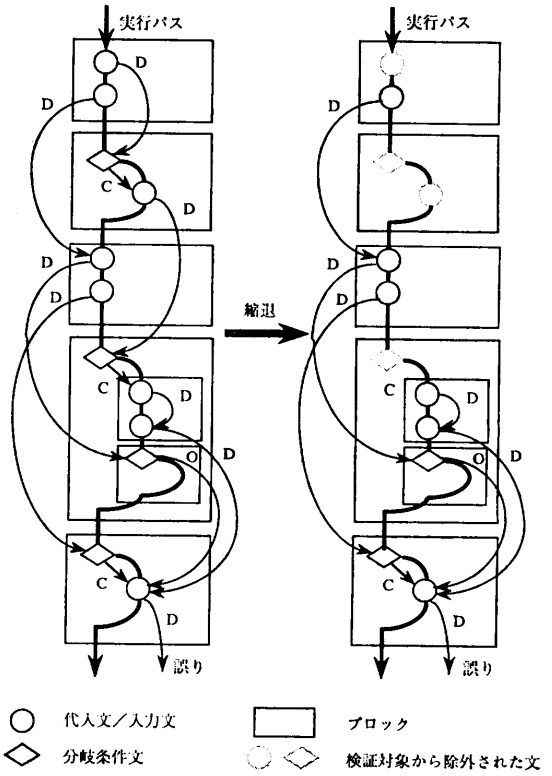


図 4 縮退された Critical フロー
Fig. 4 Reduced critical flow.

(b) Critical ブロック内の Critical フローが1つのノード r で構成される場合には，次のいずれかのバグがある（**図5**）．

(b-1) ある q が存在して，$r \in D(q, w)$ となる時．

(b-1.1) 文 $S(r)$ に名前記述誤りバグ，文記述誤りバグ，あるいは，文記述過多バグがある．

(b-1.2) $V(r)$ の中に，値の誤っている変数 u があり，Critical ブロックの直前から実行時点 r までの間に，変数 u に値を設定する処理が漏れている．すなわち，文記述漏れバグ，あるいは，名前記述誤りバグがある．

(b-1.3) 実行時点 r から Critical ブロックの直後までの間に，変数 w に値を設定する処理が漏れている．すなわち，文記述漏れバグ，あるいは，名前記述誤りバグがある．

(b-2) ある q が存在して，$r \in O(q, w)$ となる時．

(b-2.1) 文 $S(r)$ に文記述誤りバグがある．

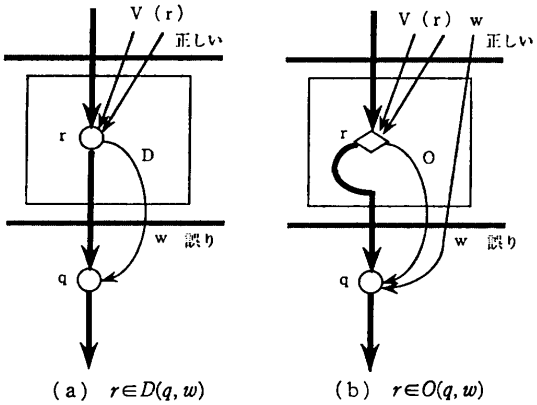
(b-2.2) $V(r)$ の中に，値の誤っている変数 u があり，Critical ブロックの直前から実行時点 r までの間に，変数 u に値を設定する処理が漏れている．すなわち，文記述漏れバグ，あるいは，名前記述誤りバグがある．

(b-2.3) Critical ブロック内に変数 w に値を設定する処理が漏れている．すなわち，文記述漏れバグ，あるいは，名前記述誤りバグがある．

図2のプログラムにおいて，変数値エラーに対するバグ究明戦略を適用した場合の例を**図6**に示す．まず，文 16 を含むブロックの直後で分割すると，フローデータは変数 c と変数 v である．Critical フローの最適分割方法についてここでは詳細にふれることはできないが，一般にノード数の多い Critical フローから判定したほうが効率が良い．そこで，変数 c の Critical フローのほうがノード数が多いため，変数 c の値の正誤を先に判定する（図6(a)）．もし，ここで変数 c の値が誤っていれば，変数 c に対する Critical フローを取り出す．ブロックを分解し，文 12 の分岐文の制御移行結果を判定する．それが正しければ，文13の分岐文ブロックの直後で分割し，フローデータ b の値の正誤を判定する（図6(b)）．変数 b の値が誤っている場合には，次のいずれかのバグがあることになる．

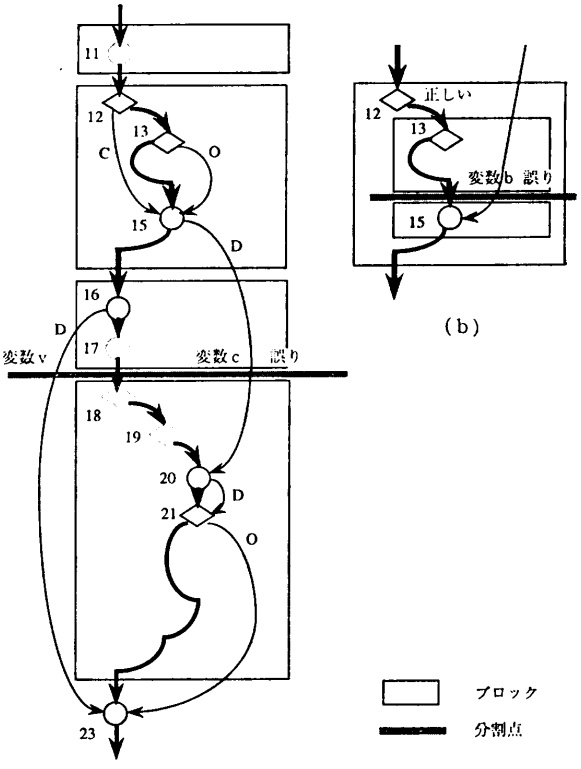
1) 文13に文記述誤りバグがある．

2) 変数 j の値が誤っており，文13までに変数 j に値を設定する処理が漏れている．



(a) $r \in D(q, w)$　　(b) $r \in O(q, w)$

図5 Critical フローが1ノードで構成される場合
Fig. 5 Critical flow consisting of one node.



(a)

図6 変数値エラーに対するバグ究明戦略
(a) 文16を含むブロックの直後で分割，(b) 文13を含む分岐文ブロックの直後で分割
Fig. 6 Bug-locating strategy for variable value errors.
(a) Division after block including 16,
(b) Division after branch block 13.

3) 文 15 の直前までに変数 b に値を設定する処理が漏れている.

4. 考　　察

変数値エラーにおいて，誤っている値に影響している文を限定する方式として，Static Slice[10)~14)]，Dynamic Slice[15)~17)]，PELAS[18)~20)]，および，本論文で提案した Critical Slice がある．2.3 節(4)に示したように，これらの間には，

Dynamic Slice⊂Critical Slice⊂PELAS
⊂Static Slice

という関係がある.

Dynamic Slice は最小の集合であるが，誤っている値に影響している，一部の文を見逃してしまう．Critical Slice はバグ潜在域であり，エラーを引き起こす可能性のある文を含む最小の集合である.

システムのガイドに従ってバグを究明する方法としては，Algorithmic Debugging[9)]，PELAS[18)~20)]，および，本論文で提案した Critical Slice 分割方式がある．副作用のある一般の手続き型プログラミング言語の場合には，関数の入力パラメータ値を基にプログラマが出力パラメータ値の正誤を判断するのは非常に困難である．たとえ，関数内で参照・設定しているグローバル変数の変数名とそれらの値をプログラマに提示しても，正しく判断できるとは限らない．なぜなら，グローバル変数の値を参照する，あるいは，設定する文の文記述漏れバグがあるかもしれないからである.

(例) グローバル変数の値を参照する文の文記述漏れバグがある場合

```
procedure P(x : out integer; y : in integer) is
begin
  x := y+Z; (x := y+Z+W; が正しいとする)
end;
```

x の値が 30 であった場合，y の値 10 と，参照しているグローバル変数 Z の値 20 を提示すると，プログラマはこの関数は正しく動作していると誤って判断しやすい.

したがって，Algorithmic Debugging をそのまま適用することはできない．また，1 つの文は繰り返し，実行されることがあるので，その文だけを見て，その文の実行時点における制御フローが正しいかどうかを判定することはできない．したがって，PELAS のように Slice を 1 つずつ逆に追跡していくのはプログラマにとっては非常に困難な作業であると思われる.

しかし，以下では，1) 一般の手続き型プログラミング言語にも，Algorithmic Debugging の方法を適用する，2) 各分割において判定するデータの数は 1 つであると仮定して，それらのバグ究明コストを比較してみる.

実行系列内の文の数を N とする．1 つの関数が呼び出している関数の数の平均を n とする．Algorithmic Debugging[9)] では，$n/2$ 回の判定で呼び出している関数の中から誤りのあるものを見つけることができるため，$n/2$ 回の判定で調査対象範囲は $1/n$ となる．したがって，判定回数は $(n/2)\log_n N$ となる．Critical Slice 分割方式では，Critical Slice の数を $M(M<N)$ とすると，分割するごとに Critical フローは縮退されていくので，判定回数は $\log_2 M$ 以下となる．PELAS では，Critical Slice より大きな集合 M' $(M'>M)$ を順に調べていくため，判定回数は $M'/2$ となる．したがって，

Critical Slice 分割方式 $\leq \log_2 M < (n/2)\log_n N$
$=$ Algorithmic Debugging
Critical Slice 分割方式 $\leq \log_2 M < M'/2$
$=$ PELAS

となる．この結果より，$M=1,000$ 程度の場合には，Critical Slice 分割方式は，PELAS 方式に比較して判定回数を数十分の 1 に減らすことができる.

5. おわりに

Critical Slice に基づく，決定性のバグ究明戦略について述べた．本論文では，エラーを引き起こす可能性のある文を含む最小の集合であるバグ潜在域を定義し，変数値エラーの場合には Critical Slice がバグ潜在域になることを示した．また，この Critical Slice を分割しフローデータの値の正誤を判定することによりバグを究明できることを示した．最後に，本方式が既存の類似方式と比較しても優れており，有効であることを示した．今後は，実用的なプログラムに対して本方式の評価を進めていきたい.

謝辞 本論文の投稿を勧めてくださり，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) 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).
2) Bruegge, B. and Hibbard, P.: Generalized Path Expressions: A High-Level Debugging Mechanism, *The Journal of Systems and Software*, Vol. 3, pp. 265-276 (1983).
3) Lazzerini, B. and Lopriore, L.: Abstraction Mechanisms for Event Control in Program Debugging, *IEEE Trans. Softw. Eng.*, Vol. 15, No. 7, pp. 890-901 (1989).
4) Olsson, R. A., Crawford, R. H., Ho, W. W. and Wee, C. E.: Sequential Debugging at a High Level of Abstraction, *IEEE Softw.*, pp. 27-36 (May 1991).
5) Olsson, R. A., Crawford, R. H. and Ho, W. W.: A Dataflow Approach to Event-based Debugging, *Softw. Pract. Exper.*, Vol. 21, No. 2, pp. 209-229 (1991).
6) Adams, E. and Muchnick, S. S.: Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations, *Softw. Pract. Exper.*, Vol. 16, No. 7, pp. 653-669 (1986).
7) Shimomura, T. and Isoda, S.: Linked-List Visualization for Debugging, *IEEE Softw.*, pp. 44-51 (May 1991).
8) Araki, K., Furukawa, A. and Cheng, J.: A General Framework for Debugging, *IEEE Softw.*, pp. 14-20 (May 1991).
9) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).
10) Weiser, M.: Programmers Use Slices When Debugging, *CACM*, Vol. 25, No. 7, pp. 446-452 (1982).
11) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 4, pp. 352-357 (1984).
12) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26-60 (1990).
13) Weiser, M. and Lyle, J.: *Experiments on Slicing-Based Debugging Aids, Empirical Studies of Programmers*, pp. 187-197, Ablex Publishing Corporation (1986).
14) Lyle, J. R. and Weiser, M.: Automatic Program Bug Location by Program Slicing, *The Second International Conference on Computers and Applications*, pp. 877-883 (June 1987).
15) Korel, B. and Laski, J.: Dynamic Program Slicing, *Inf. Process. Lett.*, Vol. 29, No. 10, pp. 155-163 (1988).
16) Agrawal, H. and Horgan, J. R.: Dynamic Program Slicing, *ACM SIGPLAN Notices*, Vol. 25, No. 6, pp. 246-256 (1990).
17) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *J. Syst. Softw.*, Vol. 13, pp. 187-195 (1990).
18) 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).
19) Korel, B.: PELAS—Program Error-Locating Assistant System, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 9, pp. 1253-1260 (1988).
20) Korel, B. and Laski, J.: Algorithmic Software Fault Localization, *Proc. 24th Annual Hawaii International Conference on System Science*, pp. 246-252 (1991).

付録 1 定理 2 の証明

(a) まず，Critical Slice 内のいずれの文に文記述誤りバグ，名前記述誤りバグ，文記述過多バグ（これらのバグを ENX バグと総称する）があっても，発見された変数値エラーを引き起こす可能性があることを示す．

$CS=CS(t,v)=\{t_1<t_2<\cdots<t_m\}$ とする．

$S(t_k)$ $(1\leq k\leq m)$ に ENX バグがあると，実行時点 t における制御フローを変えてしまう，あるいは，変数 v の値を変えてしまう可能性がある (A)

ことを数学的帰納法を用いて示す．

1) まず，$k=m$ の場合を考える．$t_m\in D(t,v)\cup O(t,v)$ ならば，$S(t_m)$ に ENX バグがあると，変数 v の値を変えてしまう可能性がある．$t_m\in C(t)$ ならば，$S(t_m)$ に ENX バグがあると，実行時点 t における制御フローを変えてしまう可能性がある．

2) 次に，$k>i$ の時には，(A)が成立しているとして，$k=i$ の場合を考える．$t_i<t_j<t$ となる，ある t_j が存在して，$t_i\in D(t_j,V(t_j))\cup O(t_j,V(t_j))$ の時には，$S(t_i)$ に ENX バグがあると，実行時点 t_j で参照している変数の値を変える可能性がある．したがって，$S(t_j)$ は正しく実行されず，実行時点 t における制御フローを変えてしまう，あるいは，変数 v の値を変えてしまう可能性がある．$t_i\in C(t_j)$ の時には，$p,q\in CS\cup\{t\}$，$i<p<q\leq t$，$t_i\in C(p)$，$t_i\notin C(q)$，$p\in D(q,V(q))$ となる，ある p,q が存在する．$S(t_i)$ に ENX バグがあると，$S(p)$ は実行されないため，$V(q)$ の値を変える可能性がある．したがって，$q=t$ ならば，変数 v の

値を変えてしまう可能性がある．$q \neq t$ ならば，$S(q)$ は正しく実行されず，実行時点 t における制御フローを変えてしまう，あるいは，変数 v の値を変えてしまう可能性がある．

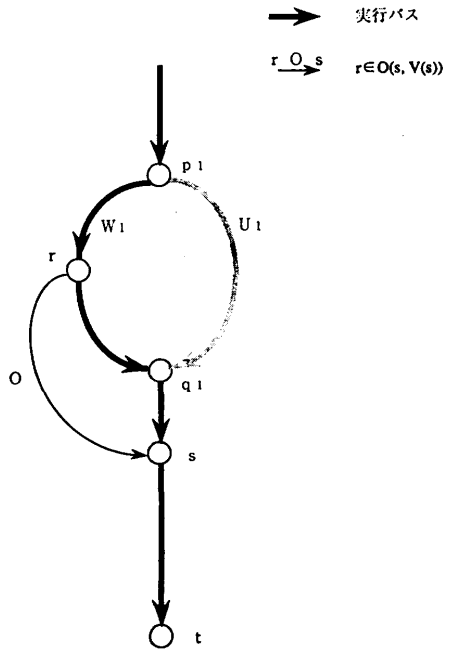
（b） 次に，Critical Slice 以外の文については，いずれの文に文記述誤りバグ，文記述過多バグ（これらのバグを EX バグと総称する）があっても，発見されたエラーを引き起こすことはないことを示す．すなわち，実行時点 t における制御フローは変らず，変数 v の値も変らないことを示す．

Critical Slice 以外の文に EX バグがあったために，途中で実行パスが変った場合を考える．$p_1 < t$, $p_1 \notin CS(t, v)$ となる $p_1 \in EX$ があって，$S(p_1)$ の EX バグのために，この時点において初めて実行パスが変ったとする．この時，実行されなかったパスを U_1 とする（**付図 1**）．

［補題 1］ U_1 は，必ず，$q_1 \leq t$ となる，ある実行時点 q_1 に戻る．すなわち，U_1 が実行される場合には，U_1 の実行の後，q_1 が実行される．

（証明） なぜなら，もし，戻らないとすると，$p_1 \in C(t)$ となり，$p_1 \notin CS(t, v)$ に矛盾するため．（証明終了）

［補題 2］ $p_1 < i < q_1$, $i \in C(t)$ となる $i \in EX$ は存在しない．



付図 1 実行パスの変化
Appendix fig. 1 Change of execution path.

（証明） もし，存在するとすると，$p_1 \in C(i)$ であり，したがって，$p_1 \in C(t)$ となり，$p_1 \notin CS(t, v)$ に矛盾．（証明終了）

［補題 3］ EX 内の誤って実行されたパスを W_1 とする．$W_1 \cap CS(t, v) \neq \phi$ の場合を考える．$r \in W_1 \cap CS(t, v)$, $q_1 \leq s$, $R_{CS}(r, s)$ ならば，$r \in O(s, V(s))$ である．

（証明） $r \in D(s, V(s))$ とすると，$p_1 \in CS(t, v)$ となり，矛盾．$r \in C(s)$ とすると，$p_1 \in C(r)$ であるから，$p_1 \in C(s)$ となり，矛盾．（証明終了）

［補題 4］ 次に実行パスが変った実行時点を p_2 とする．$s \in CS(t, v)$, $s \notin W_1$, $s \leq p_2$ ならば，実行パスが W_1 から U_1 に変っても，$S(s)$ の実行結果は変らない．ただし，次に実行パスが変った実行時点が t より前に存在しない場合には，$s \in CS(t, v)$, $s \notin W_1$, $s < t$ ならば，実行パスが W_1 から U_1 に変っても，$S(s)$ の実行結果は変らない．

（証明） 実行時点 s において，$V(s)$ に含まれる変数の値が変らなければ，実行結果は変らない．

$C = \{s \mid s \in CS(t, v), s \notin W_1, s \leq p_2$ （あるいは $s < t)\} = \{s_1 < s_2 < \cdots < s_m\}$ とおく．

$1 \leq k \leq m$ となる任意の k について，$S(s_k)$ の実行結果が変らないことを数学的帰納法を用いて示す．

1) $k = 1$ の時，$x \in D(s_1, V(s_1))$ となる x は存在しないため，$S(s_1)$ の実行結果は変らない．

2) $k < i$ の時には，$S(s_k)$ の実行結果が変らないとする．$k = i$ の時，$r \in D(s_i, V(s_i))$, $r < p_1$, $q_1 \leq s_i$ の場合を考える．

パス U_1 では $V(s_i)$ に含まれる変数を定義しない．なぜなら，もし，定義すると仮定すると，$p_1 \in O(s_i, V(s_i))$ となり，矛盾．したがって，実行時点 s_i において，$V(s_i)$ に含まれる変数の値は変らない．（証明終了）

［補題 5］ Critical Slice 以外の文の文記述誤りバグ，文記述過多バグのために，実行されなかったパスの集合を $\{U_1, U_2, \cdots, U_m\}$，誤って実行されたパスの集合を $\{W_1, W_2, \cdots, W_m\}$，実行パスを変えた実行時点の集合を $\{p_1 < p_2 < \cdots < p_m\}$，実行されなかったパスが EX に戻る実行時点を $\{q_1 < q_2 < \cdots < q_m\}$ とする．この時，$1 \leq k \leq m$ となる，任意の k について，以下のことが成立する．

1) $p_k \notin CS(t, v)$.

2) $p_k < i < q_k$, $i \in C(t)$ となる $i \in EX$ は存在しない．

3) $r \in W_k \cap CS(t, v)$, $q_k \leq s$, $Rcs(r, s)$ ならば，$r \in O(s, V(s))$ である．

4) $s \in CS(t, v)$, $s \notin W_1 \cup W_2 \cup \cdots \cup W_k$, $s \leq p_{k+1}$ ($k < m$ の時)，$s < t$ ($k = m$ の時) ならば，実行パスが変っても，$S(s)$ の実行結果は変らない．

(証明)

$k=1$ の時，補題 2, 3, 4 より，1), 2), 3), 4) は成立．$k<j$ の時，1), 2), 3), 4) が成立するとする．

$k=j$ の時，

1) $p_j \notin CS(t, v)$ である．なぜなら，$p_j \in CS(t, v)$ とすると，$S(p_j)$ の実行結果は変らないため，矛盾．

2) $p_j \notin CS(t, v)$ であるため，補題 2 の証明より，$p_j < i < q_j$, $i \in C(t)$ となる $i \in EX$ は存在しないことは明らか．

3) 補題 3 の証明より，$r \in W_j \cap CS(t, v)$, $q_j \leq s$, $Rcs(r, s)$ ならば，$r \in O(s, V(s))$ であることは明らか．

4) 補題 4 の証明より，$s \in CS(t, v)$, $s \notin W_1 \cup W_2 \cdots \cup W_j$, $s \leq p_{j+1}$ ($j < m$ の時)，$s < t$ ($j = m$ の時) ならば，実行パスが変っても，$S(s)$ の実行結果は変らないことは明らか．(証明終了)

補題 5 より，Critical Slice 以外の文については，いずれの文に文記述誤りバグ，文記述過多バグがあっても，$C(t)$ 内の各実行時点における制御移行は変らない．したがって，実行時点 t における制御フローは変らない．$D(t, v)$ となる実行時点を r とすると，r は必ず実行され，その実行結果は変らない．かつ，補題 4 の証明から，r から t に至るパスでは v を定義しないことが分かる．したがって，変数 v の値は変らない．((b)の証明終了)

付録 2 定理 3 の証明

(a) 実行時点 i 以前の実行においてバグがないとする．すなわち，実行時点 i 以前の実行において文記述漏れバグ，文記述過多バグ，文記述誤りバグ，および，名前記述誤りバグがないとする．実行時点 i 以前の実行はすべて正しいため，i の実行直後に値の誤った変数があることに矛盾する．

(b) 実行時点 i より後の実行においてバグがないとする．すなわち，実行時点 i より後の実行において文記述漏れバグ，文記述過多バグ，文記述誤りバグ，および，名前記述誤りバグがないとする．この時，たとえ，プログラムが正しく動作しても同じ変数値エラーが発生するため，矛盾することを示す．

実行時点 i の直後における制御フローは正しいため，実行時点 i 以前におけるバグのために，実行時点 i より後において実行パスが変ったとする．実行パスが変った時点を p とする．p は Critical Slice に含まれない．なぜなら，もし，p が Critical Slice に含まれるとすると，$F(i)$ 内のすべての変数の値が正しいため，矛盾する．定理 2 の補題 5 から，Critical Slice に含まれない実行時点で実行パスが変っても，変数値エラーを引き起こした時点における制御フローは変らず，その変数の値も変らないことが分かる．(証明終了)

(平成 3 年 8 月 13 日受付)
(平成 4 年 1 月 17 日採録)

下村 隆夫 (正会員)

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．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．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．