Critical Slice の拡張と分割検証の定式化

下村 隆夫†

システムのガイドに従ってバグを究明する従来の Algorithmic Debugging 手法では，手続き型言語には適用できない，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，文の記述漏れに関するバグは検出できない等の問題点があった．これに対して，手続き型言語を対象として，Critical Slice を用いた効率の良いバグ究明方式が提案されている．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る．本論文では，この Critical Slice を配列，ポインタを含むプログラムにも適用できるように拡張する．また，実行された文の間の依存関係の解析を容易にする Critical-Flow グラフを導入することにより，文の記述漏れを含む，任意のバグを究明する分割検証手順を定式化する．

Extension of Critical Slice and Formulation of Verification-by-Division

Takao Shimomura†

In the conventional algorithmic debugging methods that locate a bug under the guidance of a system, there are some problems such that they cannot be applied to procedural languages, can determine only a faulty function, that is, cannot locate a faulty statement, or cannot detect a bug concerning omitted statements. An effective bug-locating strategy for procedural languages based on critical slices has been presented. A critical slice is the minimum set containing statements that might have caused an error. This paper extends this critical slice so that it can be applied to programs including arrays and pointers. It also formulate the procedure of verification-by-division for locating any bug including omission by introducing a critical-flow graph which facilitates the analysis of relationship between executed statements.

1. はじめに

システムのガイドに従ってバグを究明する Algorithmic Debugging には，Shapiro[1]，PRESET[2]，GADT[3),4)]，PELAS[5)~7)] 等がある．Shapiro，PRESET では関数型/論理型言語を対象とし，プログラマはシステムから提示された関数の正誤を，入出力パラメータの値を基に判定する．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．この方式では，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．また，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．GADT は，この方式を手続き型言語にも適用しようという試みである．グローバル変数を参照するためのパラメータを関数に追加し，副作用のない同値な関数型プログラムに変換してからバグの究明を行う．また，Static Slicing[8)]を利用することにより，値の誤っている出力パラメータに関係する関数を特定している．しかし，グローバル変数をパラメータで渡すようにプログラムを変換しても，グローバル変数の参照漏れや設定漏れは検出できないという問題が残る．また，この方式でも，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．一方，PELAS では，手続き型言語を対象とし，実行した文の間の依存関係を実行順とは逆向きに順に調べていくことにより，バグを含む文を限定することができる．しかし，文の記述漏れ等のバグは検出できないという問題がある[8)]．

筆者は，手続き型言語を対象として，変数値エラーに対する Critical Slice に基づいた効率の良いバグ究明戦略を提案した[9)]．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，文の記述漏れを含む，任意のバグを究明することができる．

本論文では，この Critical Slice を配列，ポインタを含むプログラムにも適用できるように拡張し，実行された文の間の依存関係の解析を容易にする Critical-

† NTT ソフトウェア研究所
NTT Software Laboratories

Flow グラフを導入することにより，Critical Slice に基づいてバグを究明する分割検証手順を定式化する．

2. Critical Slice の概要

まず，文献9)で提案された Critical Slice について，概説する．

2.1 プログラムのモデル

プログラムは，代入文，分岐文，ループ文，手続き呼び出し文，および入出力文からなるとする．手続き呼び出しでは，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．

文の例を以下に示す．

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;
手続き呼び出し文	P(A, B);
入出力文	get(X); put(X);

2.2 バグの分類

プログラムの文に関するバグは，次の4つに分類できる．手続き呼び出し文において，アーギュメントの数や型を誤った場合は，Ada 言語のようにコンパイル時に検出済みとし，ここでは対象としない．

（1） 文記述漏れバグ
文の記述が漏れている場合

（2） 文記述過多バグ
余分な文の記述がある場合

（3） 文記述誤りバグ
a）代入文において，右辺の式の記述を誤った場合，b）分岐文，ループ文において，条件式の記述を誤った場合，c）手続き呼び出し文において，in パラメータに対応するアーギュメントの記述を誤った場合，d）出力文において出力アーギュメントの記述を誤った場合．

（4） 名前記述誤りバグ
a）代入文において，左辺の変数の名前を誤った場合，b）手続き呼び出し文において，呼び出す手続きの名前を誤った場合，c）手続き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，d）入力文において入力変数の名前を誤った場合．

本論文では，文記述誤りバグと文記述過多バグを総称して**値誤りバグ**，文記述漏れバグと名前記述誤りバグを総称して**設定漏れバグ**とよぶこととする．

2.3 Critical Slice の定義

代入文，入力文，出力文，分岐文の条件式部分，および，ループ文の条件式部分を，おのおの，**代入命令**，**入力命令**，**出力命令**，**分岐命令**，**ループ命令**とよぶこととする．手続き呼び出し文では，**call 命令**を実行することにより，手続きに制御を渡すものと考える．例えば，手続き P が “procedure P (X : in… ; Y : out…);” と宣言されている場合には，手続き呼び出し文 “P(A, B);” を実行すると，以下の処理を行うものと考える．

```
call P;
X:=A;
  (手続き P の本体部の実行)
B:=Y;
  (手続き呼び出し文の直後に戻る)
```

ある入力データを与えてプログラムを実行した場合，実行されたパス（命令の列）を実行系列とよぶ．t 番目に命令の実行が行われた時点を実行時点 t とよぶ．実行系列における実行時点の集合を EP で表す．実行時点 t に関して，以下の記号を定義する．

Ins(t)　t 番目に実行された命令．

Def(t)　実行時点 t における命令 Ins(t) の実行で定義された（値が設定された）変数の集合．

Use(t)　実行時点 t における命令 Ins(t) の実行で使用された（値が参照された）変数の集合．

本論文では，実行された命令の間の依存関係の解析を容易にする Critical-Flow グラフを導入するため，Critical Slice を，以下に示す4つの依存関係 Def, Ctl, CtlDef, OmsCond を用いて定義する．

（1） Def(t, v) の定義 (Definition)

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 Def(t, v) を次のように定義する．

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

以降の記述では，特に誤解がない限り，Def(t, v) が表す実行時点からなる集合 {Def(t, v)} も，Def(t, v) と表すこととする．

(2) Ctl(t) の定義 (Control)

まず，命令Aの実行の有無を決定する命令の集合CtlEx(A) を以下のように定義する．

1) if C then S1 else S2 end if; 命令AがS1，あるいは S2 部分に記述されている場合には，分岐命令Cを CtlEx(A) に含める．
2) while L loop S3 end loop; 命令AがS3部分に記述されている場合には，ループ命令Lを CtlEx(A) に含める．また，ループ命令L自身は，L∈CtlEx(L) とする．
3) procedure P(…) is begin S4 end; 命令AがS4部分に記述されている場合には，手続きPの call 命令を CtlEx(A) に含める．

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命令を実行した実行時点の集合 Ctl(t) を，次のように定義する．

1) $\max\{i \in EP \mid i<t, Ins(i) \in CtlEx(Ins(t))$, ただし，Ins(i) が call 命令の場合には，呼び出された手続きの実行が実行時点tより前に完了していない$\} \in Ctl(t)$.
2) $i \in Ctl(t)$ ならば，$\max\{j \in EP \mid j<i, Ins(j) \in CtlEx(Ins(i))$, ただし，Ins(j) が call 命令の場合には，呼び出された手続きの実行が実行時点iより前に完了していない$\} \in Ctl(t)$.

(3) CtlDef(t, v) の定義 (Control-Definition)

実行時点t，変数vに対して，実行時点の集合 CtlDef(t, v) を次のように定義する．

$$CtlDef(t, v)=Ctl(Def(t, v))-Ctl(t).$$

CtlDef(t, v) は，実行された分岐命令，ループ命令，あるいは，手続き呼び出し命令の実行時点の集合であり，その命令の実行により変数vの値を設定する命令が実行されることになり，その値が実行時点tにおける変数vの値を決定しているという性質をもつ．

(4) OmsCond(t, v) の定義 (Omission-Conditional)

実行時点t，変数vに対して，実行時点の集合 OmsCond(t, v) を次のように定義する．

$OmsCond(t, v)=\{j \in EP \mid Def(t, v)<j<t, j \notin Ctl(t)$, かつ，Ins(j) は分岐命令あるいはループ命令であり，その制御移行が変われば，その分岐文あるいはループ文内で変数vを定義する可能性がある$\}$.

OmsCond(t, v) は，実行された分岐命令，あるいはループ命令の実行時点の集合であり，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という性質をもつ．

依存関係 R=Def, CtlDef, OmsCond において，実行時点t，変数の集合Vに対して，R(t, V) を次のように定義する．

$$R(t, V)=\bigcup_{v \in V} R(t, v).$$

(5) Critical 実行時点集合 CriticalEP(t, v) の定義

実行時点tにおける，変数vに関する Critical 実行時点集合 CriticalEP(t, v) を次のように定義する．ただし，ここでは，AffectUse(i)=Use(i) とする．

1) $i \in Def(t, v) \cup OmsCond(t, v) \cup CtlDef(t, v) \cup Ctl(t)$ ならば，$i \in CriticalEP(t, v)$.
2) $i \in CriticalEP(t, v)$, かつ，$j \in Def(i, AffectUse(i)) \cup OmsCond(i, AffectUse(i)) \cup CtlDef(i, AffectUse(i))$ ならば，$j \in CriticalEP(t, v)$.

Critical 実行時点集合の要素を Critical 実行時点とよぶ．

(6) CriticalSlice(t, v) の定義

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

$$CriticalSlice(t, v)=\{Ins(j) \mid j \in CriticalEP(t, v)\}$$

CriticalSlice(t, v) は，実行時点の集合 CriticalEP(t, v) において実行された命令の集合である．Critical Slice は命令の集合であるが，Static Slice[10)~14)]や Dynamic Slice[15)~17)] のように実行可能な部分プログラムとなることを意図していない．

2.4 Critical Slice の性質

発見された変数値エラーに対して，次の2つの条件を満たす，プログラム内の命令の集合Xを，その変数値エラーに関する**値誤りバグ潜在域**とよぶ．

1) 集合X内の命令については，いずれの命令に値誤りバグがあっても，発見された変数値エラーを引き起こす可能性がある．
2) 集合X以外の命令については，いずれの命令に値誤りバグがあっても，発見された変数エラーを引き起こすことはない．すなわち，集合X以外の命令の値誤りバグを修正しても，同じ変数値エラーが発生する． □

文献9)では，以下の定理を示した．

〈定理1〉 Critical Slice

実行時点tにおいて，変数vの値が誤っている変数値エラーでは，実行時点tにおける変数vに関する Critical Slice が，その変数値エラーに関する値誤りバグ潜在域となる． □

```
1  get(x, y);
2  max := x;
3  min := x;
4  if x > y then   (正しくは、x<y)
5    max := y;
   else
6    min := y;
   end if;
7  put(max, min);
```

図 1 サンプルプログラム
Fig. 1 Sample program.

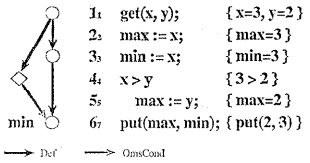


図 2 実行系列
Fig. 2 Execution sequence.

2つの値 x, y の max, min を求めるプログラムを**図 1** に示す．このプログラムに入力 x=3, y=2 を与えて実行させたときの実行系列を**図 2** に示す．図 2 において，j_s は実行時点 j において命令 S が実行されたことを表す．このとき，実行時点 6 において変数 min の値 3 が誤っている（正しくは，2）．CriticalSlice(6, min)={1, 3, 4} である．この例からわかるように，CriticalSlice 内の命令に値誤りバグがあると，変数 min に誤った値を生成してしまう可能性がある．一方，CriticalSlice 以外の命令 (2, 5, 6) に値誤りバグがあっても，変数 min には同じ値が生成される．

3. Critical Slice の拡張

本論文では，Critical Slice を配列，ポインタを含むプログラムにも適用できるように拡張する．ここでは，説明を簡単にするため，配列は 1 次元配列とする．配列要素への代入文において，添字式の記述を誤った場合は，文記述誤りバグとする．配列の添字式の範囲制約オーバは Ada 言語のように実行時に変数値エラーとして検出できるものとする．したがって，配列要素 a[m] への値の設定が，別の変数 b に対する値の設定になるという状況は対象としない．

3.1 配列への拡張

(1) OmsArray(t, v) の定義 (Omission-Array)

実行時点 t，変数 v に対して，実行時点の集合 OmsArray(t, v) を次のように定義する．

1) 変数 v がスカラ変数であるとき，OmsArray(t, v)=ϕ.
2) 変数 v が配列要素 a[m]（配列 a の m 番目の要素）であるとき，OmsArray(t, v)={j | i=Def(t, v), i<j<t, ある k が存在して，a[k]∈Def(j)}

OmsArray(t, v) は，変数 v が配列要素 a[m] である場合には，a[m] を定義した場所の実行時点より後で，同じ配列 a のある要素を定義した実行時点の集合である．

（例） 次のプログラムを実行した場合には，Def(6, a[3])=4 であり，OmsArray(6, a[3])={5} となる．

```
1  i:=1;
2  j:=1;
3  k:=3;
4  a[k]:=x;
5  a[j+1]:=y;
6  z:=a[i+2];
```

(2) Oms(t, v) の定義 (Omission)

実行時点 t，変数 v に対して，実行時点の集合 Oms(t, v) を次のように定義する．

$$Oms(t, v)=OmsCond(t, v)\cup OmsArray(t, v).$$

実行時点 t，変数の集合 V に対して，

$$Oms(t, V)=\bigcup_{v\in V} Oms(t, v).$$

(3) CriticalSlice の拡張

まず，配列要素への代入文の実行時点 j に対して，変数の集合 IndexVars(j) を次のように定める．

IndexVars(j)={w | 実行時点 j における配列要素への代入文 Ins(j) の実行において，変数 w は，左辺の配列要素の添字式の中で使用された変数である}.

例えば，命令 Ins(j) が「a[x+y−1]:=z+2;」の場合には，IndexVars(j)={x, y} である．

次に，実行時点 t における，変数 v に関する Critical 実行時点集合 CriticalEP(t, v) を次のように定義する．ただし，ここでは，実行時点 i がある実行時点 k，配列要素 a[m] に対して，i∈OmsArray(k, a[m]) である場合には AffectUse(i)=IndexVars(i)，それ以外の場合には AffectUse(i)=Use(i) とする．

1) i∈Def(t, v)∪Oms(t, v)∪CtlDef(t, v)∪Ctl(t) ならば，i∈CriticalEP(t, v).
2) i∈CriticalEP(t, v)，かつ，j∈Def(i, AffectUse(i))∪Oms(i, AffectUse(i))∪CtlDef(i, AffectUse(i)) ならば，j∈CriticalEP(t, v).

AffectUse(i) は，直感的には，実行時点 t における変数 v の値に影響を与える可能性のある，実行時点 i において命令 Ins(i) の実行で使用された変数の集合である．

CriticalSlice(t, v) は，Critical 実行時点集合 Criti-

calEP(t, v) より得られる (2.3節(6)参照).

3.1節(1)の例に示したプログラムでは, Critical-Slice(6, a[3])={1, 2, 3, 4, 5} である. したがって, 命令2, 5に値誤りバグがあると, 変数zの値に影響を与えることがわかる.

3.2 ポインタへの拡張

アロケータ (new) によりデータが動的に生成され, そのアドレスがポインタに設定されるものとする.

p :=new データ型;

ポインタは配列の変形として扱うことができる. すなわち, プログラム内で, ポインタpによって修飾されたデータ a (p→a) が参照されている場合には, pを配列aの添字と考え, a[p] が記述されているとみなして Critical Slice を求めていけばよい.

4. 分割検証の定式化

本論文では, Critical-Flow グラフを導入することにより, Critical Slice に基づいてバグを究明する手順 (分割検証手順) を定式化する.

4.1 フローデータ

Critical 実行時点集合 CriticalEP(t, v) に対して, 実行時点 i (1≦i≦t) における, **フローデータ集合 FlowData(i)** を次のように定義する.

FlowData(i)={x|r<i≦s となる, ある r, s∈CriticalEP(t, v) が存在して, x∈AffectUse(s), r=Def(s, x)}.

フローデータ集合 FlowData(i) に含まれる変数をフローデータとよぶ. x∈FlowData(i) とは, 直感的には, 実行時点 i より前に存在する, ある Critical 実行時点で変数 x に値が設定され, 実行時点 i 以後に存在する, ある Critical 実行時点で, その値が参照されていることを意味する. すなわち, FlowData(i) は, 実行時点 i を横切って, 流れるデータ (変数) の集合である.

文献9)では, 以下の定理を証明し, 制御フローの正しいある実行時点で Critical 実行時点集合を分割し, その分割点におけるフローデータの値の正誤を判定することにより, バグの存在範囲を限定することができることを示した.

〈定理2〉 フローデータ

実行時点 t において変数 v に関する変数値エラーが発生しているとする. CriticalEP(t, v) をある実行時点 i の直前で分割する. 分割点 i におけるフローデータ集合を FlowData(i) とする. 実行時点 i の直前における制御フローが正しい (すなわち, Ctl(i) 内の各実行時点における制御移行が正しい) 場合, 次のことが成立する.

分割点 i の直前において,

(a) FlowData(i) 内のある変数の値が誤っていれば, 分割点 i より前にバグが存在する.

(b) FlowData(i) 内のすべての変数の値が正しければ, 分割点 i 以後にバグが存在する. □

4.2 Critical-Flow グラフの導入

以下では, 実行時点 t において変数 v に関する変数値エラーが発生しているとする.

(1) 関係 $R_{CF}(g, h)$ の定義

実行時点 g, h∈CriticalEP(t, v)∪{t} (1≦g<h≦t) に対して,

1) h=t ならば, g∈Def(t, v)∪Oms(t, v)∪CtlDef(t, v)∪Ctl(t) のとき,

2) h<t ならば, g∈Def(h, AffectUse(h))∪Oms(h, AffectUse(h))∪CtlDef(h, AffectUse(h))のとき,

「関係 $R_{CF}(g, h)$ をもつ」という.

CriticalEP(t, v)∪{t} 内の実行時点をノード, 実行時点の間の関係 $R_{CF}(g, h)$ をアーク g→h とすることにより定義される有向グラフを, 実行時点 t における変数 v に関する Critical-Flow グラフとよぶ (図2参照).

また, 実行時点 g, h∈CriticalEP(t, v)∪{t} (1≦g<h≦t) に対して, ある変数 w が存在して, g∈Def(h, w)∪Oms(h, w)∪CtlDef(h, w) のとき, 「関係 $R_{CF/w}(g, h)$ をもつ」という.

(2) CriticalFlow(j, w) の定義

実行時点 j (1≦j≦t) における変数 w に関する Critical Flow, CriticalFlow(j, w) を次のように定義する.

1) $R_{CF/w}(h, j)$ ならば, h∈CriticaFlow(j, w).

2) h∈CriticalFlow(j, w), かつ, $R_{CF}(g, h)$ ならば, g∈CriticalFlow(j, w).

CriticalFlow(j, w)⊆CriticalEP(j, w) という関係にある. また, 分岐命令, ループ命令, あるいは, call 命令を実行した実行時点 j に対して, **制御範囲 CtlRange(j)** を次のように定義する.

CtlRange(j)={k∈EP|j∈Ctl(k)}.

〈定理3〉 Critical Flow

実行時点 t における変数 v に関する Critical-Flow グラフにおいて, 実行時点 j (1<j≦t), 変数 w に対して, r<j≦s, w∈AffectUse(s), r=Def(s, w) となる r,

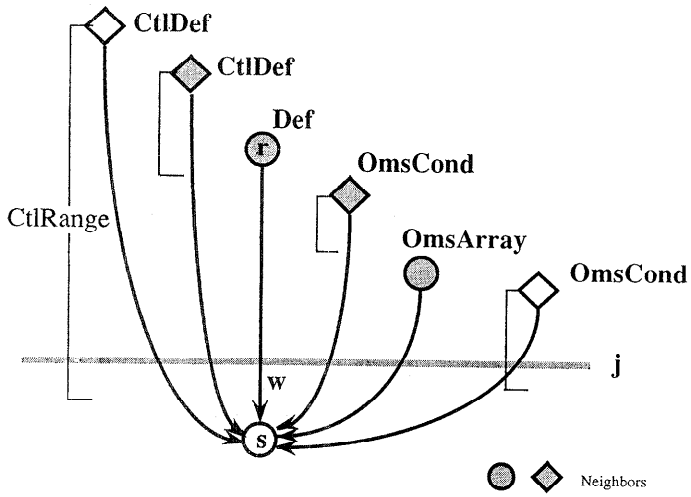


図 3 CriticalFlow (j, w) の求め方
Fig. 3 Acquisition of CriticalFlow (j, w).

$s \in CriticalEP(t, v) \cup \{t\}$ が存在するとする．Neighbors $= \{m \in CriticalEP(t, v) \mid m < j,\ m \notin Ctl(j),\ R_{CF/w}(m, s)\}$ とおく．

このとき，Critical-Flow グラフにおいて，Neighbors に含まれる．いずれかのノードに到達するノードの集合を ReachableNodes とおくと，CriticalFlow (j, w)＝ReachableNodes となる（**図 3**）．

（証明）

$ReachableNodes \subseteq CriticalFlow(j, w)$ である．$Def(j, w) = r = Def(s, w)$ であるから，$h \in Oms(j, w) \cup CtlDef(j, w)$ ならば，$h \in Neighbors$ となる．したがって，$CriticalFlow(j, w) \subseteq ReachableNodes$ である．□

4.3 分割検証

（1） Critical 実行時点の除去

Critical 実行時点集合 CriticalEP(t, v) に含まれない分岐命令，ループ命令，あるいは，call 命令の制御移行が誤っている場合について考察する．Ctl(i) に含まれる実行時点を **Ctl(i)-実行時点**とよぶこととする．

〈定理 4〉 Critical 実行時点の除去

分割点 $i(1 \leq i \leq t)$ に対して，Critical 実行時点集合 CriticalEP(t, v) に含まれない Ctl(i)-実行時点 j の実行結果（制御移行）が誤っているとする．以下の記号を定義する．

$Inner = CriticalEP(t, v) \cap CtlRange(j)$.

$Affect = \{e \subset CriticalEP(t, v) \mid e < j$, かつ，ある実行時点の列 $\langle f_1, f_2, \cdots, f_m \rangle$ が存在して，$f_1 = e$ かつ，$1 \leq r < m$ となる任意の r に対して，$f_{r+1} \notin CtlRange(j)$, $R_{CF}(f_r, f_{r+1})$, かつ，$f_m \in Def(t, v) \cup Oms(t, v) \cup CtlDef(t, v) \cup Ctl(t)\}$.

$Upper = \{u \in CriticalEP(t, v) \mid u < j\} - Affect$.

$NoAffect = Upper \cup Inner$.

このとき，NoAffect 内の実行時点で実行した命令の値誤りバグのために，それらの実行結果が変わったとしても，現在発生している変数値エラーの原因とはならない．すなわち，NoAffect 内の実行時点，および，実行時点 j が正しく実行されても，同じ変数値エラーが発生する（**図 4**）．

（証明）

実行時点 j は Critical 実行時点ではないため，実行時点 j でパスが変わっても，変数値エラーに影響を与えない（文献 9) の補題 5 参照）．また，実行時点 j が正しく実行されると制御移行が変わるため，Inner 内の実行時点は実行されない．したがって，実行されない Inner 内の実行時点のみに影響を与えていた，Upper 内の実行時点における実行結果が変わっても，やはり変数値エラーに影響を与えない．したがって，同じ変数値エラーが発生する． □

（2） 分割検証アルゴリズム

実行時点 k に対して，Ctl(k)-実行時点の制御移行が正しく，かつ，実行時点 k の直前において，フローデータ集合 FlowData(k) 内のすべての変数の値が正

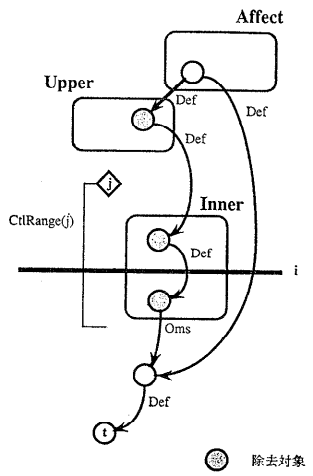


図 4 Critical 実行時点の除去
Fig. 4 Removal of critical execution points.

```
while not (検証対象の Critical 実行時点の数=0 or k+1=j) loop
  分割点 i を決める.
  分割点 i までの実行系列の正誤をプログラマが判定する.
  if Critical 実行時点に含まれる Ctl(i)-命令の制御移行が誤っている then
    if call 命令の制御移行が誤っている then
      call 命令の名前記述誤りバグ.
    elsif 分岐命令，ループ命令で使用した変数wの値が誤っている then
      前方除去分割.
    elsif 分岐命令，ループ命令で使用した変数の値はすべて正しい then
      分岐命令，ループ命令の文記述誤りバグ.
    end if;
  elsif Critical 実行時点に含まれない Ctl(i)-命令の制御移行が誤っている then
    内部除去分割.
  elsif Ctl(i)-命令の制御移行はすべて正しい then
    フローデータ集合 FlowData(i) の値の正誤をプログラマが判定する.
    if フローデータwの値が誤っている then
      前方除去分割.
    elsif フローデータの値はすべて正しい then
      後方除去分割.
    end if;
  end if;
end loop;

if 検証対象の Critical 実行時点の数=0 then
  フロー正常点以後でフロー異常点より前に，設定漏れバグが存在する.
elsif 検証対象の Critical 実行時点の数=1 then
  命令 Ino(k) のバグ（文記述誤り，名前記述誤り，あるいは文記述過多バグ）である.
end if;
```

図 5 分割検証アルゴリズム
Fig. 5 Algorithm for verification-by-division.

しいとき，この実行時点kを**フロー正常点**とよぶ．分割検証開始時は，プログラムの実行開始時点をフロー正常点とみなす．実行時点jに対して，Ctl(j)-実行時点の制御移行が正しく，かつ，実行時点jの直前において，ある変数wの値が誤っているとき，この実行時点jを**フロー異常点**とよぶ．

実行時点tにおいて変数vに関する変数値エラーが発生している場合に，Critical 実行時点集合 Critical EP(t, v) の分割を繰り返すことにより，バグを究明するアルゴリズムを**図5**に示す．フロー正常点をk，フロー異常点をjとする．分割点iは，Critical 実行時点の数を二分する実行時点にとることとする．ただし，フロー正常点k以後でフロー異常点jより前にCritical 実行時点sが唯1つしか存在しない場合には，Critical 実行時点sの直前，および，直後で分割する．

実行系列の正誤の判定では，実行系列の実行順序が正しいかどうかを判定する．すなわち，Ctl(i)-実行時点に対して，その分岐命令，ループ命令，あるいは，call 命令の制御移行が正しいかどうかを判定する．Ctl(i)-実行時点で実行された命令を **Ctl(i)-命令**とよぶこととする．

分割検証アルゴリズムでは，以下の3種類の分割を用いる．現在の分割検証において検証対象である Critical 実行時点の集合を CurrentVerifiedEP（最初の分割では，CurrentVerifiedEP=CriticalEP(t, v)），次の分割検証で検証対象とする Critical 実行時点の集合を NextVerified EP とする．

1) 前方除去分割

実行時点jにおいて変数wの値が誤っているとする．

NextVerifiedEP=CriticalFlow(j, w)∩

```
 1  get(n, a);
 2  s:=a[1];
 3  i:=2;
 4  while i≦n loop
 5    s:=s−a[i];     (正しくは, s:=s+a[i];)
 6    i:=i+1;
    end loop;
 7  if s<10 then     (正しくは, s>10)
 8    if s mod 2≠0 then
 9      s:=s+1;
      end if;
    end if;
10  put(s);
```

図 6 プログラム example
Fig. 6 Program example.

CurrentVerifiedEP.

2) 後方除去分割
Next VerifiedEP＝{p∈CurentVerifiedEP| i≦p}.

3) 内部除去分割
定理4で定義した記号を用いると，
NextVerifiedEP＝Curent VerifiedEP－NoAffect.

前方除去分割における，Critical Flow(j, w) は，Critical Flow 定理（定理3）より，Current VeifiedEP で構成される Critical-Flow グラフから容易に求めることができる．内部除去分割を行った場合には，次の分割検証では，分割点は，制御移行の誤っている Ctl(i)-実行時点 j の制御範囲 CtlRange(j) 外にとる．

Critical 実行時点集合 CriticalEP (t, v) に含まれない Ctl(i)-実行時点の制御移行が誤っている場合，この原因を追跡することは，現在問題となっている変数値エラー（実行時点 t における変数 v の値の誤り）とは別のエラーの原因を追跡することになる（ただし，バグは異なる場合も，同一である場合もありうる）．ここでは，分割検証によってバグの存在範囲をフロー正常点からフロー異常点までの範囲内に限定できているので，その中のバグを見逃さないことが重要である．

（3） 分割検証の結果

分割検証を繰り返すと，最後には，フロー正常点以後でフロー異常点より前に存在する検証対象の Critical 実行時点の数が1以下となる．フローデータ定理（定理2）より，バグはフロー正常点以後でフロー異常点より前に存在するが，その範囲内に検証対象の Critical 実行時点が存在しない場合には，Critical Slice 定理（定理1）より，設定漏れバグがあることがわかる．検証対象の Critical 実行時点 k が唯1つ存在し，k がフロー正常点，k＋1 がフロー異常点であれば，命令 Ins(k) にバグ（文記述誤り，名前記述誤り，あるいは文記述過多バグ）がある．

5. バグ究明例

分割検証によるバグ究明例を，以下に2例示す．1例は内部除去分割を行う例であり，もう1つは Oms-

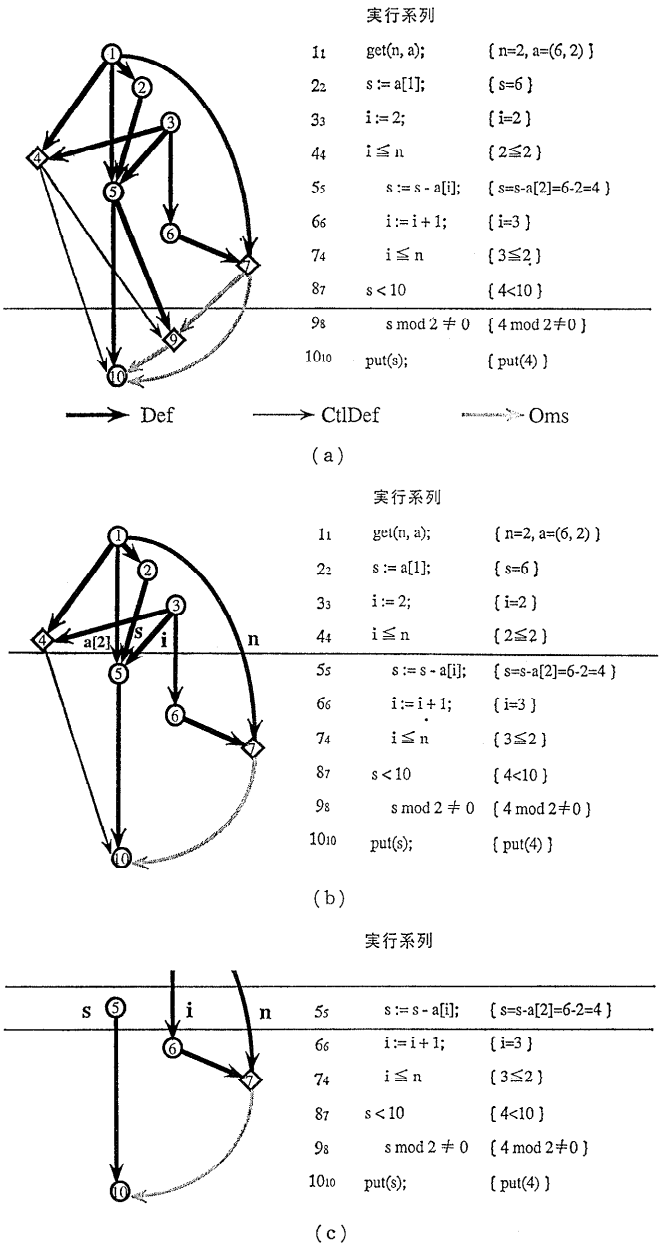


図7 プログラム example のバグ究明
Fig. 7 Bug localization of program example.

Array 依存関係を用いる例である．分割点は，説明の都合上，必ずしも，最適な分割点とはなっていない．

(1) プログラム example のバグ究明例

サンプルプログラム example を**図6**に示す．プログラム example は，配列 a から要素の値を n 個入力し，変数 s の値を出力する．変数 s の値は，以下の式で定義される．

$$sum = \sum_{i=1}^{n} a[i]$$

s=sum+1 (sum>10，かつ，sum が奇数のとき)

s=sum (それ以外のとき)

プログラム example に，入力 n=2, a=(6, 2) を与えて実行すると，出力した変数 s の値 4 が誤っている（正しくは，8）．その実行系列と Critical-Flow グラフを**図7**(a)に示す．

1) まず，実行時点の 9 の直前で実行系列を分割する（図7(a)）．実行時点 8 の制御移行が誤っている．実行時点 8 は Critical 実行時点集合に含まれないため，内部除去分割を行い，実行時点 8 の制御範囲 CtlRange(8) に含まれる実行時点 9 を検証対象から除外する．
2) 次に，実行時点 5 の直前で分割する（図7(b)）．実行時点 4 の制御移行は正しい．フローデータは変数 s, a[2], i, n であり，それらの値 s=6, a[2]=2, i=2, n=2 は正しいので，後方除去分割を行う．
3) 最後に，実行時点 6 の直前で分割する（図7(c)）．フローデータは変数 s, i, n である．Critical Flow に含まれる要素数が最も多い，変数 s の値を最初に調べると，s=4 であり，誤っている．したがって，命令 5「s :=s−a[i];」にバグがあることがわかる．なお，命令 7 は，現在発生している変数値エラーには関係しない別のバグである．

```
11  while sp_op>0 loop
12    if op[sp_op]='+' then
13      val[sp_val-1]:=val[sp_val-1]+val[sp_val];
      else
14      val[sp_val-1]:=val[sp_val-1]*val[sp_val];
      end if;
15    sp_val:=sp_op+1; (正しくは, sp_val:=sp_val-1;)
16    sp_op:=sp_op-1;
    end loop;
17  ans:=val[1];
18  put(ans);
```

図8 プログラム calculator の一部
Fig. 8 Part of Program calculator.

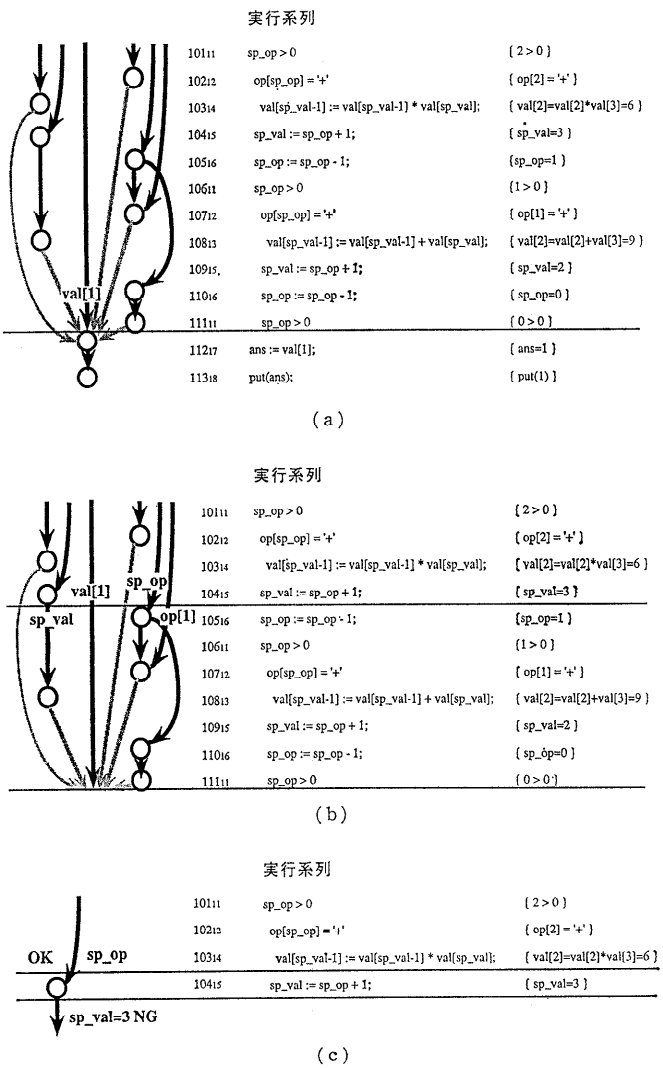


図9 プログラム calculator のバグ究明
Fig. 9 Bug Localization of Program calculator.

(2) プログラム calculator のバグ究明例

式の値を計算するプログラム calculator の一部を**図8**に示す．命令 11 を実行するときには，各変数には以下の値が正しく設定されているとする．

sp_val=3, val[1]=1, val[2]=2, val[3]=3

sp_op=2, op[1]='+', op[2]='*'

プログラム calculator は，変数 sp_val, sp_op, 配列 val, op の値を基に，変数 ans の値を出力する．変数 ans の値は，以下の式で定義される．

ans=val[1]op[1](val[2]op[2]val[3])

=1+2*3

プログラム calculator を実行すると，出力した変数 ans の値1が誤っている（正しくは，7）．その実行系列と Critical-Flow グラフを**図9**(a)に示す．

1) まず，実行時点112の直前で実行系列を分割する（図9(a)）．フローデータは変数 val[1] であり，その値 val[1]=1 は誤っているので，前方除去分割を行う．
2) 次に，実行時点105の直前で分割する（図9(b)）．実行時点101の制御移行は正しい．フローデータは変数sp_val, val[1], sp_op, op[1] である．CriticalFlow に含まれる要素数が最も多い，変数 sp_val の値を最初に調べると，sp_val=3 であり，誤っているため，前方除去分割を行う．
3) 最後に，実行時点104の直前で分割する（図9(c)）．フローデータは変数 sp_op であり，その値 sp_op=2 は正しい．したがって，命令15「sp_val :=sp_op+1;」にバグがあることがわかる．

6. おわりに

配列，ポインタを含む手続き型言語のデバッグにおいて，Critical Slice に基づいて，一定の手順に従いバグを究明する分割検証方式について述べた．本方式では，システムが分割点までの実行系列と分割点におけるフローデータの値を提示するので，プログラマはそれらの正誤を判定するだけでよい．また，フローデータはその分割点において調べるべき必要十分な変数の集合である．今後は，実用的なプログラムに対して本方式の評価を行っていく予定である．

謝辞 本研究に対して貴重な示唆と助言をいただきました National Computer Systems 研究所の James R. Lyle 博士，および，Wayne State 大学の Bogdan Korel 博士に感謝いたします．

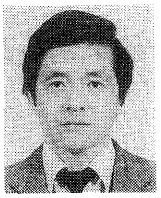
また，本研究を進めるにあたり日頃から励ましと助言をいただいています，NTT ソフトウェア研究所細谷僚一所長，後藤滋樹部長，伊藤正樹リーダに深謝いたします．

参考文献

1) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).
2) Takahashi, H. and Shibayama, E.: PRESET—A Debugging Environment for Prolog, Logic Programming Conference, Tokyo, pp. 90-99 (1985).
3) Shahmehri, N., Kamkar, M. and Fritzson, P.: Semiautomatic Bug Localization in Software Maintenance, *Proceedings of Conference on Software Maintenance*, pp. 30-36 (Nov. 1990).
4) Fritzson, P., Gyimothy, T., Kamkar, M. and Shahmehri, N.: Generalized Algorithmic Debugging and Testing, *Sigplan Notices*, Vol. 26, No. 6, pp. 317-326 (1991).
5) 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).
6) Korel, B.: PELAS—Program Error-Locating Assistant System, *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253-1260 (1988).
7) Korel, B., and Laski, J.: Algorithmic Software Fault Localization, *Proc. 24th Annual Hawaii International Conference on System Science*, pp. 246-252 (1991).
8) 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).
9) 下村隆夫：変数値エラーにおける Critical Slice に基づくバグ究明戦略，情報処理理学会論文誌，Vol. 33, No. 4, pp. 501-511 (1992).
10) Weiser, M.: Programmers Use Slices When Debugging, *CACM*, Vol. 25, No. 7, pp. 446-452 (1982).
11) Weiser, M.: Program Slicing, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 352-357 (1984).
12) 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 (1990).
13) Weiser, M. and Lyle, J.: *Experiments on Slicing-Based Debugging Aids, Empirical Studies of Programmers*, Ablex Publishing Corporation, pp. 187-197 (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, *Information Processing Letters*, Vol. 29, No. 10, pp. 155-163 (1988).
16) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *J. Systems Software*, Vol. 13, pp. 187-195 (1990).
17) Agrawal, H. and Horgan, J. R.: Dynamic program Slicing, *ACM SIGPLAN Notices*, Vol 25, No. 6, pp. 246-256 (1990).

(平成4年10月8日受付)
(平成5年9月8日採録)



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．