Vol. 35 No. 8 情報処理学会論文誌 Aug. 1994

# プログラムバグ潜在域の最適化に関する考察

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アルゴリズミックデバッギング技術では，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，手続き型言語を対象とし，プログラムバグ潜在域 Critical Slice を用いて，文の記述漏れを含むすべての種類のバグを文単位で検出できるバグ究明方式が提案されている．プログラムを実行したとき，ある変数値エラーが発生している場合に，Critical Slice は値誤りバグ（式の誤りに関するバグ）の存在範囲を示すプログラム内の命令の集合である．本論文では，実行された命令の間の依存関係に着目して，この Critical Slice のサイズを最適化する技法について提案する．

# Study of Optimization of Potential Fault Region

Takao Shimomura[†,*]

In the conventional algorithmic debugging methods that locate a fault under the guidance of a system, there are some problems such that they cannot be applied to procedural languages, can locate a faulty function but not a faulty statement, or cannot detect a fault concerning missing statements. A fault-locating method for procedural languages, which can locate all types of faults including missing statements by using critical slices, has been presented. For a variable-value error that occurred, a critical slice is a set of the statements that might have caused the error if they include wrong-value faults. This paper presents a method for optimizing the size of this critical slice by analyzing relationship between executed instructions.

## 1. はじめに

システムのガイドに従ってバグを究明する従来のアルゴリズミックデバッギング技術には，Shapiro[1]，GADT[2),3)]，PELAS[4),5)] 等がある．Shapiro 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[6] 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[6]．

これに対し，筆者は，手続き型言語を対象とし，プログラムバグ潜在域 Critical Slice を用いて，文の記述漏れを含むすべての種類のバグを文単位で検出できるバグ究明方式を提案した[7]．プログラムを実行したとき，ある変数値エラーが発生している場合に，Critical Slice は，値誤りバグ（式の誤りに関するバグ）の存在範囲を示すプログラム内の命令の集合である．この Critical Slice を分割し，分割点におけるフローデータ（分割点を横切って流れるデータ）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

† NTT ソフトウェア研究所
NTT Software Laboratories
* 現在 ATR 通信システム研究所
Presently with ATR Communication Systems Research Laboratories

意のバグを究明することができる．本論文では，実行された命令の間の依存関係に着目して，この Critical Slice のサイズを最適化する技法について提案する．

## 2. Critical Slice

本章では Critical Slice について，その概要を述べる（厳密な定義については文献8)を参照)．

（1）プログラムのモデルとバグの分類

プログラムは，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;), 入力文 (get (X);), および出力文 (put (X);) からなるとする（手続き呼び出し文があっても以降の議論は成立するが，ここでは割愛する)．プログラムは文の集合であると考えると，プログラムに含まれるバグは，文の記述が漏れている場合（文記述漏れバグ)，余分な文の記述がある場合（文記述過多バグ)，および，文の記述に誤りがある場合に分類できる．文の記述に誤りがある場合は，別の変数に値を設定するという誤りを引き起こす場合（名前記述誤りバグ）と，それ以外の場合（文記述誤りバグ）に分類できる．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**と呼ぶこととする．バグの例を**表1**に示す．

（2）実行系列

代入文，入力文，出力文を，おのおの，代入命令，入力命令，出力命令，また，分岐文，ループ文の条件式部分を，おのおの，分岐命令，ループ命令と呼ぶこととする．ある入力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）を実行系列と呼ぶ．$t$ 番目に命令の実行が行われた時点を実行時点 $t$ と呼ぶ．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．

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．

（例）二つの値 $x, y$ の最小値，最大値を求めるプログラム min_max を**図1**に示す．入力 $x=3, y=2$ を与えて，このプログラムを実行したときの実行系列を**図2**（c）に示す．

$j_S$ は実行時点 $j$ において命令 S が実行されたことを表す．

（3）実行された命令の間の依存関係

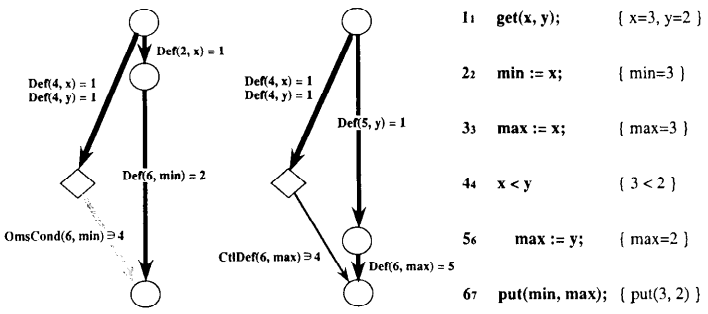
実行時点 $t$, 変数 $v$ に対して，実行された命令の間の五つの依存関係，Def (t, v), Ctl (t), CtlDef (t, v), OmsCond (t, v), OmsArray (t, v) を定義する．**図3**はこれらの依存関係を説明している．命令 "v :=" は変数 $v$ を定義する（値を設定する）ことを，命令 ":=

**表1** バグの例
Table 1 Fault examples.

| バグの種類 | 正しい文 | 誤った文 |
|---|---|---|
| 値誤りバグ | | |
| 文記述誤りバグ | X := Y+Z; | X := Y−W; |
| | A[m+1]:= X; | A[n−1]:= X; |
| | if X>Y then | if X<Z then |
| 文記述過多バグ | | X := Y+Z; |
| 設定漏れバグ | | |
| 文記述漏れバグ | X := Y+Z; | |
| 名前記述誤りバグ | X := Y+Z; | W := Y+Z; |
| | A[m]:= X; | B[m]:= X; |

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

**図1** プログラム min_max
Fig. 1 Program min_max.




（a）変数 min への影響　（b）変数 max への影響　（c）実行系列

（a）Influence on min　（b）Influence on max　（c）Execution sequence

**図2** 変数 min, max への影響
Fig. 2 Influence on variables min and max.

v" は変数 $v$ を使用する（値を参照する）ことを示す．":=" はある命令を表す．◇は分岐命令あるいはループ命令の実行時点を，○はそれ以外の命令の実行時点を表している．

（a） Def(t, v) (Definition)…実行時点 $t$ より前で，変数 $v$ に最後に値を設定した実行時点である．

（b） Ctl(t) (Control)…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である．

（c） CtlDef(t, v) (Control-Definition)…次のように定義される．

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

CtlDef(t, v) は，分岐命令あるいはループ命令の実行時点の集合であり，それらの命令の制御移行により変数 $v$ の値を設定する命令が実行されることになり，かつ，そこで設定された値が実行時点 $t$ で使用している変数 $v$ の値となっているという性質をもつ．

（d） OmsCond (t, v) (Omission-Conditional)…次のように定義される．

OmsCond(t, v)＝{実行時点 j | Def(t, v)<j<t, j∉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，分岐命令あるいはループ命令の実行時点の集合であり，それらの命令の制御移行が変われば，変数 $v$ の値を設定する可能性があり，かつ，そのために，それらの命令の制御移行が変われば，実行時点 $t$ で使用している変数 $v$ の値を変えたかもしれないという性質をもつ．

（e） OmsArray(t, v) (Omission-Array)…変数 $v$ が配列要素 a[m] である場合に定義される．実行時点 $t$ より前で，かつ，配列要素 a[m] を最後に定義した実行時点より後で，同じ配列 a の別の要素を定義した実行時点の集合である．

(例) 図2に示したプログラム min_max の実行では，実行時点6において，二つの変数 min, max の値が誤っている．実行時点6を起点として，変数 min に関して依存関係を辿ると，Def(6, min)＝2, CtlDef(6, min)＝$\phi$, OmsCond(6, min)＝{4}, OmsArray(6, min)＝$\phi$ より，実行時点2，4が抽出される．次に，実行時点4から，実行時点4で使用している変数 $x$, $y$ に関して依存関係を辿ると，Def(4, x)＝1, Def(4, y)＝1 となる．実行時点2から，実行時点2で使用している変数 $x$ に関して依存関係を辿ると，Def(2, x)＝1 となる．したがって，実行時点6から変数 min に関して依存関係を辿ると，実行時点1，2，4が抽出される（図2（a））．抽出された，これらの実行時点で実行された命令は確かに実行時点6における変数 min の値に影響を与えている．一方，抽出されなかった命令3，5，6については，たとえ値誤りバグを含んでいても，実行時点6における変数 min の値には影響を与えないことがわかる．

実行時点6を起点として，変数 max に関して依存関係を辿ると，Def(6, max)＝5, CtlDef(6, max)＝{4}, OmsCond(6, max)＝$\phi$, OmsArray(6, max)＝$\phi$ より，実行時点4，5が抽出される．次に，実行時点5から，実行時点5で使用している変数 $y$ に関して依存関係を辿ると，Def(5, y)＝1 となる．実行時点4から，実行時点4で使用している変数 $x$, $y$ に関して依存関係を

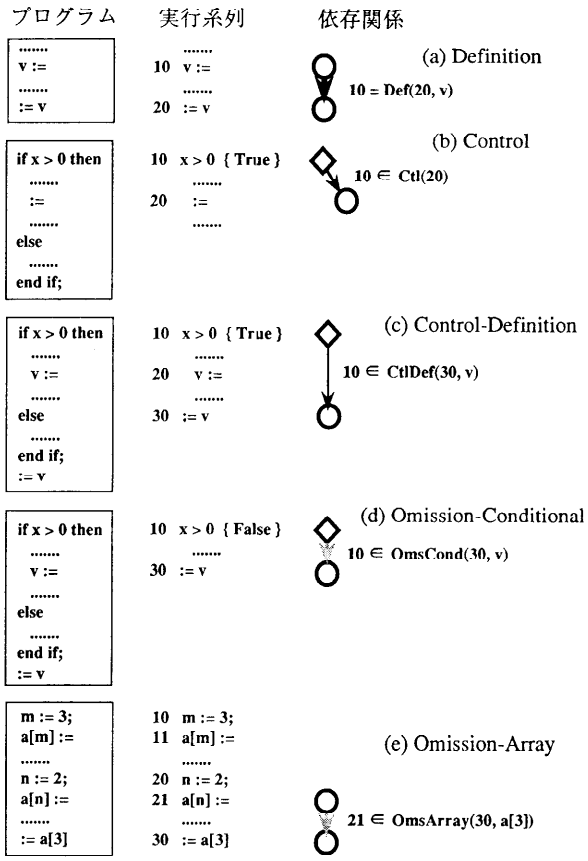



図 3 実行された命令の間の依存関係
Fig. 3 Dependences between executed instructions.

辿ると，Def(4, x)=1, Def(4, y)=1 となる．したがって，実行時点6から変数 max に関して依存関係を辿ると，実行時点 1，4，5 が抽出される（図2（b））．抽出された，これらの実行時点1，4，5で実行された命令 1，4，6 は確かに実行時点6における変数 max の値に影響を与えている．一方，抽出されなかった命令2，3，5については，たとえ値誤りバグを含んでいても，実行時点6における変数 max の値には影響を与えないことがわかる．

（4）Critical Slice

図2（a），（b）で示したように，ある実行時点を起点として，ある変数に関して，実行時点の間の依存関係 Def, CtlDef, OmsCond, OmsArray を次々と辿ることにより得られる有向グラフ（ノード：実行時点，アーク：実行時点の間の依存関係）を **Critical-Flow グラフ**と呼ぶ．

起点以外の実行時点 $i$ から依存関係を辿る場合には，その実行時点 $i$ において使用されている各変数 $w \in$Use(i) に対して，Def(i, w)，CtlDef(i, w) OmsCond(i, w) あるいは OmsArray(i, w) となる実行時点を辿っていく．ただし，図3に示した実行時点 21 ∈OmsArray(30, a[ 3 ]) のように，配列要素への代入命令を実行した実行時点であり，ある実行時点（ここでは 30）に対して OmsArray 依存関係にある場合には，その配列要素の添え字式の中で使用された変数（ここでは $n$）に対して，次の依存関係を辿っていく．実行時点 $i$ において，辿る対象となる変数の集合を AffectUse(i) と表すこととする．

CriticalEP(t, v) は，実行時点 $t$ から変数 $v$ に関して，および，Ctl(t)に含まれる各実行時点 $s$ から Use(s) に含まれる変数に関して，依存関係を辿ったときに得られる Critical-Flow グラフ内の実行時点の集合を表す．CriticalEP(t, v) 内の要素を Critical 実行時点と呼ぶ．実行時点 $t$ における変数 $v$ に関する Critical Slice＝CriticalSlice(t, v) は CriticalEP(t, v) 内の各実行時点で実行された命令の集合である．

## 3. プログラムバグ潜在域

ここでは，変数値エラーが発生している場合に値誤りバグの存在範囲を示すプログラム内の命令の集合であるプログラムバグ潜在域について考察する．

（1）変数値エラー

実行時点 $t$ における**制御フロー**が正しいとは，Ctl(t) 内の各実行時点において実行された分岐命令あるいはループ命令の制御移行が正しい場合をいう．

次のいずれかの条件が成立する場合に，実行時点 $t$ において変数 $v$ に関する**変数値エラー**が発生しているという．

（a）実行時点 $t$ における制御フローは正しいが，実行時点 $t$ の直前における変数 $v$ の値が誤っている．

（b）実行時点 $t$ における制御フローが誤っている．

□

実行時点 $t$ における変数 $v$ に関する変数値エラーを VarValErr(t, v) と表すこととする．

（2）プログラムバグ潜在域

プログラムPにある入力Iを与えて実行したとき，発見された変数値エラー E＝VarValErr(t, v) に対して，次の条件を満たす．プログラムP内の命令の部分集合 P′ を，その変数値エラーEに関する**プログラムバグ潜在域**と呼ぶ．

> 集合 P′ に含まれない命令の値誤りバグは，変数値エラーEの原因とはならない．すなわち，集合 P′ に含まれない命令の値誤りバグを修正しても，変数値エラーEを修正することはできない．□

この定義によれば，もとのプログラムP自身もプログラムバグ潜在域の一つになる．自明でないプログラムバグ潜在域が，次の定理で示す Critical Slice である．

[Critical Slice 定理]

Critical Slice＝CriticalSlice(t, v) は，変数値エラー E＝VarValErr(t, v) に関するプログラムバグ潜在域である．

（証明）

Critical Slice に含まれない命令の値誤りバグを修正してプログラムを実行した場合を考える．このとき，変数値エラーEを修正することはできないことを示す．すなわち，Ctl(t) 内の各実行時点に制御が到達するならば，それらの実行時点における制御移行結果は変わらず，かつ，実行時点 $t$ に制御が到達するならば，その実行時点における変数 $v$ の値も変わらないことを示す．

実行時点 $t$ までの実行パスに関して，以下の三つの場合に分けられる．

(Case 1) 実行されるパスが変わらない．

(Case 2) 実行されるパスが変わり，変わったパスの中で実行されたループ文（あるいは手続き）の実行が停止しない．

(Case 3) 実行されるパスは変わったが，変わったパス

の中で実行された文の実行は完了している．

Case 1 の場合には，実行時点 $s$ を Critical 実行時点（すなわち，$s \in$CriticalEP(t, v)）とすると，実行時点 $s$ において，命令 Ins(s) の実行結果は変わらない．なぜなら，命令 Ins(s) は Critical Slice に含まれるため変更されていない，かつ，実行パスは変わっていないため，実行時点 $s$ における命令 Ins(s) で使用された変数の値を定義した実行時点も必ず実行され，その命令の実行結果も変わらないため．したがって，実行時点 $t$ における制御フローも変わらず，実行時点 $t$ の直前における変数 $v$ の値も変わらない．

Case 3 の場合には，初めて実行パスが変わった実行時点を $p_1$ とすると，Critical 実行時点における実行結果は変わらないことから，$p_1 \notin$CriticalEP(t, v) である．$p_1 \notin$Ctl(t) より，実行パスは，$q_1 \leqq t$ となるある実行時点 $q_1$ に戻る（**図 4**）．また，$p_1 \notin$Ctl(t) より，$p_1 < i < q_1$, $i \in$Ctl(t) となる実行時点 $i$ は存在しない．次に実行パスが変わった実行時点を $p_2$ とする．$s_1 \in$ CriticalEP(t, v), $q_1 \leqq s_1 \leqq p_2$ となる最初の実行時点を $s_1$ とする．実行パスが変わっても，実行時点 $s_1$ における命令 Ins($s_1$) の実行結果は変わらない．なぜなら，命令 Ins($s_1$) は Critical Slice に含まれるため変更されていない，かつ，実行時点 $s_1$ における命令 Ins($s_1$) で使用された任意の変数を $w, d =$Def($s_1$, w) とすると，$p_1 \notin$CtlDef($s_1$, w) より $d < p_1$ であり実行時点 $d$ における命令 Ins(d) の実行結果は変わらない，かつ，

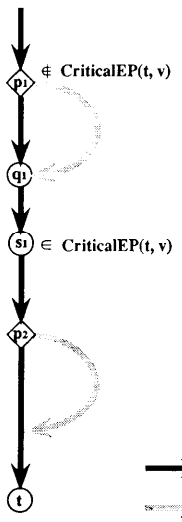



**図 4** 値誤りバグを修正したために変わった実行パス
Fig. 4 Changed execution path because of modification of wrong-value faults.

$p_1 \notin$OmsCond($s_1$, w) より変わったパスの中で変数 $w$ が定義されることはないため．同様にして，実行されるパスが変わっても，Ctl(t) 内の実行時点は実行され，その制御移行は変わらない．したがって，実行時点 $t$ も実行され，実行時点 $t$ の直前における変数 $v$ の値も変わらないことを示すことができる（文献 8）参照）．

Case 2 の場合には，実行パスは Ctl(t) 以外の実行時点において変わり，その変わったパスの中で実行が停止しないため，やはり，変数値エラーEを修正することはできない． □

（3）最小のプログラムバグ潜在域

Critical Slice は自明でない一つのプログラムバグ潜在域であるが，バグを究明する観点からは，プログラムバグ潜在域はできるだけ小さいほうが良い．しかし，最小のプログラムバグ潜在域というものは存在しない．**図 5** に示すプログラム prog1 を実行したときの，実行時点 7 における変数 $V$ に関する Critical-Flow グラフについて考えてみる（**図 6**）．

命令 “A := 1;”，命令 “B := 1;” は CriticalSlice (7, V) に含まれるが，変数 $A, B$ の中，どちらか一方の値がどのように変わっても，実行時点 7 における変数 $V$ の値には影響を与えない．したがって，命令 “A := 1;”，命令 “B := 1;” の中，どちらか一方の代入

```
1  A := 1;
2  B := 1;
3  X := A**2+B**2;
4  V := 0;
5  if X≧1 then
6     V := 1;
   end if;
7  put (V);
```

**図 5** プログラム prog 1
Fig. 5 Program prog 1.

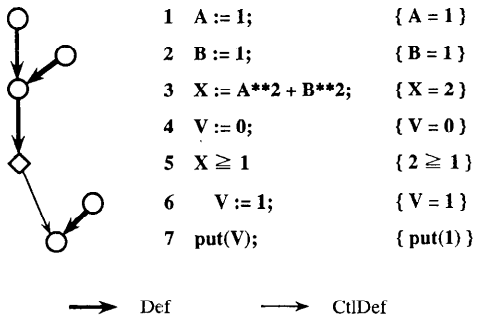



**図 6** プログラム prog 1 の Critical-Flow グラフ
Fig. 6 Critical-flow graph of program prog 1.

命令を CriticalSlice (7, V)= {1, 2, 3, 5, 6} から除外したものはプログラムバグ潜在域となる．しかし，両方の代入命令を除外することはできない．なぜなら，"A := 0;", "B := 0;" とすると，$X=0<1$ となり，変数 $V$ の値が 0 に変わってしまうため．したがって，最小のプログラムバグ潜在域というものは存在しない．

## 4. プログラムバグ潜在域の最適化理論

最小のプログラムバグ潜在域というものは存在しない．そこで，変数値エラーに影響を与えない命令を実行している実行時点を Critical-Flow グラフから除外することにより，極小となるプログラムバグ潜在域を求めるアルゴリズムについて考察する．

Critical-Flow グラフ内の各ノード $j \in$CriticalEP (t, v) ($j<t$) に対して，実行時点の大きなものから順に，以下の処理を行う．

(Step 1) 実行時点 $j$，すでに Critical-Flow グラフから除外された実行時点，および，Critical 実行時点でない実行時点における実行結果（代入命令において設定される値，分岐命令における分岐結果等）を任意に変えて，プログラムを実行することにより，変数値エラーに影響を与えるかどうか（すなわち，実行時点 $t$ における制御フローを変える，あるいは，実行時点 $t$ の直前における変数 $v$ の値を変えるかどうか）を調べる．ただし，実行結果は，命令の値誤りバグを修正するという方法により変えるものとする（したがって，代入命令において左辺の変数名は変えない）．変数値エラーに影響を与えなければ，その実行時点 $j$，および，実行時点 $j$ を終点とするアークを Critical-Flow グラフから除外する．

(Step 2) 出力アークの存在しないノード，および，そのノードの入力アークを除外する．

このように，Critical-Flow グラフからノードやアークを除外することを繰り返し，最後まで残った各ノード（実行時点）で実行された命令の集合をプログラムバグ潜在域とする．

上記の手続きに従ってプログラムバグ潜在域の最適化を行う場合，変数値エラーに影響を与えるかどうかを調べるためには，除外の対象とする実行時点 $j$ の実行結果を任意に変えるだけではなく，すでに Critical-Flow グラフから除外された実行時点，および，Critical 実行時点でない実行時点における実行結果も任意に変えることが重要である．**図 7** に示すプログラム prog 2 を実行したときの，実行時点 9 における変数 $V$ に関する Critical-Flow グラフについて考える（**図 8**）．

まず，Critical 実行時点である分岐命令 "$A \geqq 0$" を除外の対象とし，その実行結果だけを変えて実行してみる．すると，$B=2, X=3$ となるが，分岐命令 "$X>1$" の実行結果は変わらないため，出力される変数 $V$ の値 1 は変わらない．したがって，分岐命令 "$A \geqq 0$" の実行結果は変数値エラーに影響を与えないようにみえる．しかし，Critical 実行時点ではない代入命令 "$B$ := 2" の実行結果も変わり，$B=0$ となれば，$X=1$ となるため，分岐命令 "$X>1$" の実行結果は変わり，出力される変数 $V$ の値も 0 に変わる．したがって，分岐命令 "$A \geqq 0$" を除外することはできない．

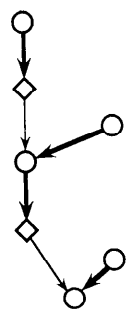
## 5. 記号実行による最適化

上記で述べたプログラムバグ潜在域の最適化を実現する一つの方法として，記号実行による方法が考えられる．

除外の対象とする実行時点 $j$，すでに Critical-Flow グラフから除外された実行時点，および，Critical 実

```
1  A := 1;
2  B := 2;
3  if A≧0 then
4      B := 1;
   end if;
5  X := B+1;
6  V := 0;
7  if X>1 then
8      V := 1;
   end if;
9  put (V);
```

**図 7** プログラム prog 2
Fig. 7 Program prog 2.



```
1  A := 1;          {A = 1}
2  B := 2;          {B = 2}
3  A ≧ 0            {1 ≧ 0}
4    B := 1;        {B = 1}
5  X := B + 1;      {X = 2}
6  V := 0;          {V = 0}
7  X > 1            {2 > 1}
8    V := 1;        {V = 1}
9  put(V);          {put(1)}
```

Def　　CtlDef

**図 8** プログラム prog 2 の Critical-Flow グラフ
Fig. 8 Critical-flow graph of program prog 2.

行時点でない実行時点における実行結果を任意に変えて記号実行を行い，変数値エラーに影響を与えるかどうかを調べる．代入命令 "X :=" の実行結果を任意に変える場合には，次に変数 $X$ が再定義されるまでの間，変数 $X$ の使用は $X1$（$X1$ は変数 $X$ の任意の値を表す）に置き換える．

図8に示したプログラム prog2 の実行では，命令 2，6 を実行した実行時点は Critical 実行時点ではない．ここで，命令 4 を除外の対象として，記号実行すると，**図9**に示すようになる．$B2 \leqq 0$ かつ $V1 \neq 1$ となる解 $B2$，$V1$ が存在するため，$V=1$ 以外の値が出力されうる．したがって，命令 4 を除外することはできない．

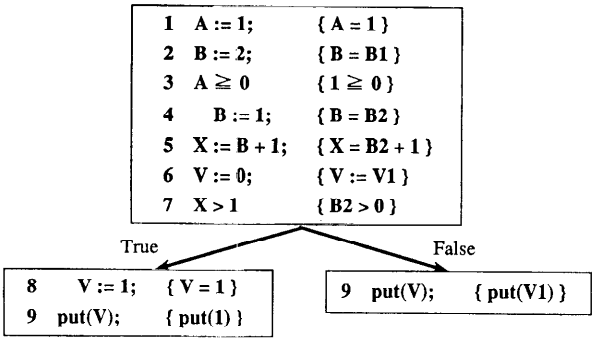



**図 9** プログラム prog 2 の記号実行
Fig. 9 Symbolic execution of program prog 2.

## 6. 依存関係に着目した最適化技法

記号実行を行った結果，実行時点 $t$ において変数 $v$ の値を変える解が見つかれば，実行時点 $j$ を除外することはできない．除外できることを示すためには，解が一つも存在しないことを示す必要がある．しかし，これは一般には困難である．記号実行はテストデータの自動生成等によく用いられているが[9),10)]，そこではパス条件を満たす一つの解を求めているだけである．解が見つからない場合にはパス条件を満たすテストデータは生成されないが，解が存在しないことを保証しているわけではない．また，この記号実行による方式は，Critical 実行時点で実行された各命令を一つ一つ順にプログラムバグ潜在域から除外する候補にして記号実行を行ってみる必要があるので効率が悪い．

そこで，本章では，命令の間の依存関係に着目したプログラムバグ潜在域の最適化技法について述べる．AffectUse(i) は，実行時点 $i$ において辿る対象となる変数の集合であり，AffectUse(i)⊆Use(i) であった（2章(4)参照）．この AffectUse(i) をできる限り小さい集合にとることによりプログラムバグ潜在域の最適化を実現する．変数 $w \in$ AffectUse(i) に関して実行時点 $i$ から依存関係を辿ることによって抽出される実行時点の集合を CriticalFlow(i, w) と表すこととする．

(1) Ctl, CtlDef, OmsCond における最適化

ある実行時点に対して Ctl, CtlDef, OmsCond 依存関係にある（すなわち，ある Critical 実行時点 $k$ が存在して，$j \in$ Ctl(k) ∪ CtlDef(k, AffectUse(k)) ∪ OmsCond(k, AffectUse(k)) となっている）分岐命令あるいはループ命令の実行時点 $j$ において，以下のように AffectUse(j) の最適化を行う．

(Opt 1) 分岐命令あるいはループ命令の条件式の値を決定した項 Term に含まれる，使用された変数の集合（これを Use(Term) で表す）を AffectUse(j) とする．

(Opt 2) 条件式の値を決定した項が複数存在する場合には，最終的に Critical-Flow グラフのサイズを最小とする項 $\text{Term}_i$ の Use($\text{Term}_i$) を AffectUse(j) とする．

(Opt 3) 条件式の値を決定した項に含まれる，使用さ

```
1  X := 1;
2  Y := 0;
3  Z := X+3;
4  W := 0;
5  if Y<1 or Z>2 then
6      W := 2;
   end if;
7  V := W+Z;
8  put (V);
```

**図 10** プログラム prog 3
Fig. 10 Program prog 3.

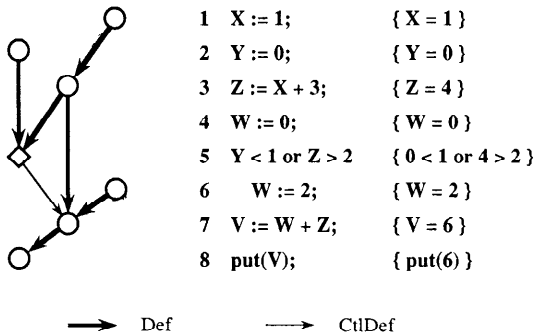



**図 11** プログラム prog 3 の Critical-Flow グラフ
Fig. 11 Critical-flow graph of program prog 3.

れた変数がその項の値を変えるような値域をもたない場合には，AffectUse(j)=$\phi$ とする．

(例) **図 10** に示すプログラム prog 3 を実行したときの，実行時点 8 における変数 $V$ に関する Critical-Flow グラフを**図 11** に示す．実行時点 5∈CtlDef(7, W) である．項 "Y<1", 項 "Z>2" はどちらも True で，分岐命令の条件式 "Y<1 or Z>2" の値 True を決定している．CriticalFlow(5, Y)= {2}, Critical-Flow (5, Z)= {1, 3} であり，項 "Y<1" に関する Critical Flow のほうが小さい．しかし，Critical-Flow (7, Z)= {1, 3} であるため，項 "Z>2" を選択したほうが (Use ("Z>2")= {Z} を AffectUse (5) とするほうが)，最終的な Critical-Flow グラフは小さくなる (Opt 2). したがって，この例では，命令 "Y := 0;" をプログラムバグ潜在域から除外することができる．

（2）OmsArray における最適化

ある Critical 実行時点 $k$ が存在して，$j$∈Oms-Array (k, a[m]) となっている配列要素 a[n] ($n \neq m$) への代入命令の実行時点 $j$ において，以下のように AffectUse (j) の最適化を行う．$d$=Def(k, a[m]) とする．

```
d   a[m] := b;
j   a[n] := c;
k   := a[m];
```

(Opt 1) $c \neq b$ となる変数 $c$ の値域が存在しなければ，AffectUse(j)=$\phi$ とする．

(Opt 2) $n=m$ となる変数 $n$ の値域が存在しなければ，AffectUse(j)=$\phi$ とする．

最適化の例を次章に示す．

## 7. 最適化技法の適用例

依存関係に着目したプログラムバグ潜在域の最適化技法について，その適用例を以下に二つ示す．

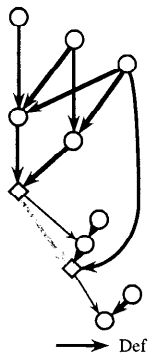
(例 1) 分岐命令における最適化

**図 12** に示すプログラム prog_cond に入力 $x=1$, $y=2$ を与えて実行した時の実行系列，および，出力文の実行時点における出力変数 $w$ に関する Critical-Flow グラフを**図 13** に示す．また，この Critical-Flow グラフを基にしてプログラムバグ潜在域の最適化を行った結果を**表 2** に示す．

(Opt 0) Critical-Flow グラフから求められる Critical 実行時点を示す．ここでは Critical 実行時点の数

```
 1  get (x);
 2  get (y);
 3  z := 1;
 4  w := 1;
 5  x := x+y+z-1;
 6  y := y-z;
 7  w := w+x;
 8  if x**2+1<0 and y>1 then
 9      z := 2;
    else
10      w := 3;
    end if;
11  x := w+1;
12  if x>4 or z<2 then
13      w := 4;
    end if;
14  put (w);
```

**図 12** プログラム prog_cond
Fig. 12 Program prog_cond.



```
1₁   get(x);                  { x = 1 }
2₂   get(y);                  { y = 2 }
3₃   z := 1;                  { z = 1 }
4₄   w := 1;                  { w = 1 }
5₅   x := x + y + z -1;       { x = 3 }
6₆   y := y - z;              { y = 1 }
7₇   w := w + x;              { w = 2 }
8₈   x**2 + 1 < 0 and y > 1   { 10 < 0 and 1 > 1 }
9₁₀     w := 3;               { w = 3 }
10₁₁ x := w + 1;              { x = 4 }
11₁₂ x > 4 or z < 2           { 4 > 4 or 1 < 2 }
12₁₃    w := 4;               { w = 4 }
13₁₄ put(w);                  { put(4) }
```

Def ⟶ CtlDef ⟿ Oms

**図 13** プログラム prog_cond の Critical-Flow グラフ
Fig. 13 Critical-flow graph of program prog_cond.

**表 2** プログラム prog_cond におけるプログラムバグ潜在域の最適化
Table 2 Optimization of potential fault region in program prog_cond.

| 実行系列 | | | Opt 0 | Opt 1 | Opt 2 | Opt 3 |
|---|---|---|---|---|---|---|
| $1_1$ | get (x); | {x=1} | ○ | ○ | × | × |
| $2_2$ | get (y); | {y=2} | ○ | ○ | ○ | × |
| $3_3$ | z := 1; | {z=1} | ○ | ○ | ○ | ○ |
| $4_4$ | w := 1; | {w=1} | × | × | × | × |
| $5_5$ | x := x+y+z-1; | {x=3} | ○ | ○ | × | × |
| $6_6$ | y := y-z; | {y=1} | ○ | ○ | ○ | × |
| $7_7$ | w := w+x; | {w=2} | × | × | × | × |
| $8_8$ | x**2+1<0 and y>1 | {10<0 and 1>1} | ○ | ○ | ○ | ○ |
| $9_{10}$ | w := 3; | {w=3} | ○ | × | × | × |
| $10_{11}$ | x := w+1; | {x=4} | ○ | × | × | × |
| $11_{12}$ | x>4 or z<2 | {4>4 or 1<2} | ○ | ○ | ○ | ○ |
| $12_{13}$ | w := 4; | {w=4} | ○ | ○ | ○ | ○ |
| $13_{14}$ | put (w); | {put (4)} | 10 | 8 | 6 | 4 |

は10であるが，Opt 1～Opt 3までの最適化により，Critical実行時点の数を4まで削減することができる．

(Opt 1) 実行時点 11 で実行された分岐命令の条件式 "x>4 or z<2" の値を決定した項 "z<2" に含まれる変数 $z$ のみを AffectUse(11) として，最適化を行った結果である．変数 $x$ に関係する，実行時点10において実行された命令 "x := w+1;"，および，実行時点9において実行された命令 "w := 3;" が除外できる．

(Opt 2) 実行時点8で実行された分岐命令 "x**2+1<0 and y>1" では，どちらの項も False となり条件式の値を決定している．CriticalFlow (8, x)= {1, 2, 3, 5}, CriticalFlow (8, y)= {2, 3, 6} である．そこで，最終的に Critical-Flow グラフのサイズを最小とする項 "y>1" を選択し，AffectUse (8)= {y} として，最適化を行った結果である．変数 $x$ に関係する，実行時点5において実行された命令 "x := x+y+z−1;"，および，実行時点1において実行された命令 "get(x);" が除外できる．

(Opt 3) 実行時点8で実行された分岐命令 "x**2+1<0 and y>1" では，項 "x**2+1<0" は False となり条件式の値を決定しているが，その値を True に変えるような変数 $x$ の値域は存在しないため，AffectUse(8)= $\phi$ として，最適化を行った結果である．最終的に，変数 $y$ に関係する，実行時点6において実行された命令 "y := y−z;"，および，実行時点2において実行された命令 "get(y);" も除外されることになる．

一方，最後まで Critical 実行時点として残った命令については，その実行結果が変わると，出力文の実行時点における出力変数 $w$ の値を変えることがわかる（**表 3**）．

**表 3** 値誤りバグの修正による値の変化
Table 3 Changes in values by modification of wrong-value faults.

| | 命　　令 | 値誤りバグ修正後の命令 | 値の変化 |
|---|---|---|---|
| prog_cond | 3 "z := 1;" | "z := 2;" | w=3 |
| | 8 "x**2+1<0 and y>1" | True | w=2 |
| | 12 "x>4 or z<2" | False | w=3 |
| | 13 "w := 4;" | "w := 1;" | w=1 |
| prog_array | 1 "x := 10;" | "x := 20;" | z=20 |
| | 3 "i := 0;" | "i := −1;" | z=30 |
| | 5 "k := 1;" | "k := 0;" | z=20 |
| | 6 "m := 1;" | "m := 2;" | z=未定義値 |
| | 8 "a[m+1] := x;" | "a[m+1] := x+10;" | z=20 |
| | 9 "a[k+2] := y;" | "a[k+1] := y;" | z=20 |
| | 10 "a[2*j+1] := x+y;" | "a[j+2] := x+y;" | z=30 |
| | 11 "a[n+1] := 10;" | "a[n] := 20;" | z=20 |

(例2) 配列要素への代入命令における最適化

**図 14** に示すプログラム prog_array を実行したときの実行系列，および，実行時点 12 における変数 $z$ に関する Critical-Flow グラフを**図 15** に示す．また，この Critical-Flow グラフを基にしてプログラムバグ潜在域の最適化を行った結果を**表 4** に示す．

(Opt 0) Critical-Flow グラフから求められる Critical 実行時点を示す．

(Opt 1) 実行時点 8=Def(12, a[2]) において，配列要素 a[2] に設定された値は 10 である．実行時点 11 ∈OmsArray(12, a[2]) では，配列要素 a[3] に値10を設定しているが，右辺には変数を含まないた

```
 1  x := 10;          {x=10}
 2  y := 20;          {y=20}
 3  i := 0;           {i=0}
 4  j := 0;           {j=0}
 5  k := 1;           {k=1}
 6  m := 1;           {m=1}
 7  n := 2;           {n=2}
 8  a[m+1] := x;        {a[2]=10}
 9  a[k+2] := y;        {a[3]=20}
10  a[2*j+1] := x+y;    {a[1]=30}
11  a[n+1] := 10;       {a[3]=10}
12  z := a[i+2];        {z=a[2]=10}
```

**図 14** プログラム prog_array
Fig. 14 Program prog_array.

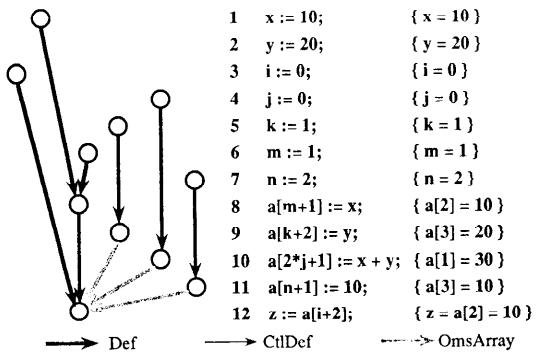



**図 15** プログラム prog_array の Critical-Flow グラフ
Fig. 15 Critical-flow graph of program prog_array.

表 4 プログラム prog_array におけるプログラムバグ潜在域の最適化
Table 4 Optimization of potential fault region in program prog_array.

| | 実行系列 | | Opt 0 | Opt 1 | Opt 2 |
|---|---|---|---|---|---|
| 1 | x := 10; | {x=10} | ○ | ○ | ○ |
| 2 | y := 20; | {y=20} | × | × | × |
| 3 | i := 0; | {i=0} | ○ | ○ | ○ |
| 4 | j := 0; | {j=0} | ○ | ○ | × |
| 5 | k := 1; | {k=1} | ○ | ○ | ○ |
| 6 | m := 1; | {m=1} | ○ | ○ | ○ |
| 7 | n := 2; | {n=2} | ○ | × | × |
| 8 | a[m+1] := x; | {a[2]=10} | ○ | ○ | ○ |
| 9 | a[k+2] := y; | {a[3]=20} | ○ | ○ | ○ |
| 10 | a[2*j+1] := x+y; | {a[1]=30} | ○ | ○ | ○ |
| 11 | a[n+1] := 10; | {a[3]=10} | ○ | ○ | ○ |
| 12 | z := a[i+2]; | {z=a[2]=10} | 10 | 9 | 8 |

め，10以外の値が設定されることはありえない．そこで AffectUse(11)=$\phi$ として，最適化を行った結果である．変数 $n$ に関係する，実行時点7において実行された命令 "n := 2;" が除外できる．

(Opt 2) 実行時点 10∈OmsArray(12, a[2]) では，配列要素 a[1] に値30を設定しているが，配列の添字式は "2*j+1" であるため，配列要素 a[2] に値が設定されることはありえない．そこで，AffectUse(10)=$\phi$ として，最適化を行った結果である．変数 $j$ に関係する，実行時点4において実行された命令 "j := 0;" が除外される．

一方，最後まで Critical 実行時点として残った命令については，その実行結果が変わると，実行時点12における変数 z の値を変えることがわかる（表3）．

## 8. おわりに

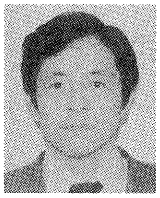
プログラムバグ潜在域の最適化問題について考察し，最適化に関する理論および最適化を行う技法について明らかにした．記号実行による最適化は，記号実行を行った結果，変数値エラーを引き起こしている変数の値が定数になり，その値が変わらない場合（例えば，図6において Critical 実行時点2を除外する場合）には有効である．依存関係に着目した最適化技法の中，分岐命令における最適化では，条件式の各項で使用している変数の集合が互いに素である場合には，項の数を $N$ とすると，それ以後に辿る Critical-Flow グラフはほぼ $1/N$ 程度に縮退できる．また，複数の分岐命令を論理演算子によって一つの分岐命令にマージすることができる場合（例えば，"if A then if B then...end if; end if;" を "if A and B then...end if;"）には，マージした分岐命令に対してこの最適化技法を適用することができる．

**謝辞** 本研究を進めるにあたり日頃から励ましと助言を頂きました，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) Shahmehri, N., Kamkar, M. and Fritzson, P.: Semi-automatic Bug Localization in Software Maintenance, *Proceedings of Conference on Software Maintenance*, pp. 30-36 (Nov. 1990).
3) Fritzson, P., Gyimothy, T., Kamkar, M. and Shahmehri, N.: Generalized Algorithmic Debugging and Testing, *SIGPLAN Notices*, Vol. 26, No. 6, pp. 317-326 (1991).
4) Korel, B.: PELAS—Program Error-Locating Assistant System, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 9, pp. 1253-1260 (1988).
5) Korel, B. and Laski, J.: Algorithmic Software Fault Localization, *Proc. 24th Annual Hawaii International Conference on System Science*, pp. 246-252 (1991).
6) 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).
7) 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，情報処理学会論文誌，Vol. 33, No. 4, pp. 501-511 (1992).
8) Shimomura, T.: Critical Slice-Based Fault Localization for Any Type of Error, *IEICE Transactions on Information and Systems*, Vol. E 76-D, No. 6, pp. 656-667 (1993).
9) Clarke, L.: A System to Generate Test Data and Symbolically Execute Programs, *IEEE Trans. Softw. Eng.*, Vol. SE-2, No. 3, pp. 215-222 (1976).
10) Howden, W.: Symbolic Testing and the DISSECT Symbolic Evaluation System, *IEEE Trans. Softw. Eng.*, Vol. SE-4, No. 4, pp. 266-278 (1977).

（平成5年7月27日受付）
（平成6年4月21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．同年日本電信電話公社（現 NTT）電気通信研究所勤務．昭和62年より NTT ソフトウェア研究所勤務．平成5年12月より ATR 通信システム研究所に出向中．平成4年4月から平成6年3月まで電気通信大学大学院情報システム学研究科客員助教授．これまでに，汎用クロスアセンブラ，統計解析プログラム，仕様記述言語，グラフィックパッケージ，CASE ツール，テストカバレッジアナライザ，ビジュアルデバッガ，情報通信システムセキュリティの研究実用化に従事．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．