ＡＴＲテクニカルレポート表紙

〔非公開〕

ＴＲ－Ｃ－０１２７

# 高信頼性ソフトウェア設計手法の研究

下村　隆夫
Takao SHIMOMURA

１９９６．２．２９

ＡＴＲ通信システム研究所

# 高信頼性ソフトウェア設計手法の研究
**Research on Design Methods of High-Quality Software**

下村　隆夫
Takao Shimomura

ATR 通信システム研究所
ATR Communication Systems Research Laboratories

１９９６年２月２９日

目次

はじめに....4

I部　副作用のないソフトウェア改造方式の研究....5
1．概要....5
2．プログラムスライシング....5
（1）フローグラフ....5
（2）スタティックスライス....6
3．従来のソフトウェア改造方式....7
（1）改造による影響を波及させない改造方式....7
（2）独立な命令に対する編集....7
（3）プログラム WordCounter の改造....7
4．非改造部分から受ける影響....11
（1）課金プログラム Charge1 とその改造内容....11
（2）改造プログラム Charge2....13
（3）改造プログラム Charge3....14
（4）改造部分が非改造部分から受ける影響....15
5．コード移動による影響の除去....15
（1）問題の定式化....15
（2）コード移動の十分条件....16
（3）コード移動アルゴリズム....18
（4）改造プログラム Charge2 におけるコード移動....20
6．コード付加による影響の除去....22
（1）改造プログラム Charge3 におけるコード移動....22
（2）コード付加アルゴリズム....22
（3）改造プログラム Charge3 におけるコード付加....24
7．考察....26
（1）スタティックスライスを用いた独立改造方式....26
（2）コード移動アルゴリズム....26
（3）非改造部分を利用した改造....27

II部　分散処理システム障害解析支援方式の研究 ........ 28
1．概要 ........ 28
2．Critical Sliceの概要 ........ 28
2．1　バグの分類 ........ 28
2．2　Critical Sliceの定義 ........ 29
2．3　Critical Sliceの性質 ........ 31
3．バグ究明方式 ........ 36
3．1　出力に関するエラー ........ 36
3．2　出力異常エラーに対するバグ究明 ........ 36
3．3　出力なし無限ループエラーに対するバグ究明 ........ 40
3．4　エラー誤認の回避 ........ 42
4．実現方式 ........ 45
4．1　処理形態 ........ 45
4．2　ユーザインタフェース ........ 47
4．3　テストフェーズの処理 ........ 48
4．4　デバッグフェーズの処理 ........ 48
4．5　分割検証に必要な情報の獲得 ........ 49

おわりに ........ 51
参考文献 ........ 53

## はじめに

通信ネットワークのオープン化，サービスの多様化に伴い，通信サービスを提供する通信処理ソフトウェアの規模は大きく，かつ，複雑になりつつある．また，ネットワークセキュリティの観点からも，品質の高いソフトウェアを提供することが重要な課題となっている．本テクニカルレポートでは，高信頼性ソフトウェア設計手法の研究の一環として行われた2つの研究「副作用のないソフトウェア改造方式の研究」，および，「分散処理システム障害解析支援方式の研究」について，それらの研究成果と今後の課題について述べる．

「I部　副作用のないソフトウェア改造方式の研究」に関しては，ソフトウェアを改造する場合，従来技術では，改造対象機能および非改造機能について計算している部分をプログラム内から抽出するスライシング技術を応用して改造作業を支援しており，非改造機能に影響を与えないように改造を行うことはできるが，逆に，非改造部分から改造部分が影響を受け，正しく改造できない場合が起こりうるという問題があった．そこでこの問題を解決するため，改造部分が非改造機能に関するスライスから受ける影響を解析し，これを除去することにより，非改造機能には影響を与えず，かつ，改造部分が非改造部分から影響を受けないように改造を行うことを可能とするソフトウェア独立改造方式を提案する．本方式により，改造対象機能に関するスライスだけに着目して改造作業を進めることができる．また，テスト作業も改造対象機能に関するスライスに対して行えばプログラム全体に対してテストしたのと同じ効果をもち，かつ，非改造機能に関する再テストも不要となる．

「II部　分散処理システム障害解析支援方式の研究」に関しては，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，変数値エラーに対して，変数値エラーを引き起した可能性のある文の集合であるCritical Slice を用いた決定性のバグ究明方式を提案する．本報告では，さらに，プログラムのテストにおいて観察される，出力異常エラー（変数値エラーを含む），出力漏れエラー，および，出力なし無限ループエラーの3つのエラーに対する Critical Slice に基づいたバグ究明方式を提案する．出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べ，そのような場合には，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができることを示す．また，本バグ究明方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについても述べる．

## I部　副作用のないソフトウェア改造方式の研究

### 1．概要

仕様変更や機能拡充のために，ソフトウェアのある部分を改造する場合，その改造が他の部分にも影響を与える可能性があるため，改造した後で，改造対象部分のテストだけでなく，改造対象外の部分が元の仕様通りに動作するかどうかを確認する再テストが必要になっている 1), 2).

そこで，ソフトウェアを改造する場合に，プログラムスライシング技術を応用して，改造対象機能および非改造機能について計算している部分をプログラム内から抽出し，非改造機能に影響を与えないように改造作業を支援する方式が提案されている 3)-6).改造対象機能に関するスライスを求めれば，プログラム内で改造対象機能について計算している部分が分かるため，改造作業が容易になる．しかし，この方式では，非改造機能に影響を与えないように改造を行うことはできるが，逆に，非改造部分から改造部分が影響を受け，正しく改造できない場合が起こりうるという問題がある．

本論文では，改造部分が非改造機能に関するスライスから受ける影響を解析し，これを除去することにより，非改造機能には影響を与えず，かつ，改造部分が非改造部分から影響を受けないように改造を行うことを可能とするソフトウェア独立改造方式を提案する．本方式により，改造対象機能に関するスライスだけに着目して，改造作業を進めることができるとともに，テスト作業もプログラム全体に対して行うのではなく，改造対象機能に関するスライスについて独立に行うことができる．

2章ではプログラムスライシング技術について概説し，3章でスタティックスライスを利用した従来のソフトウェア改造方式について述べる．また，4章で非改造部分から受ける影響について説明し，5章，6章でこの影響を自動的に除去し，改造対象機能に関するスライスだけに着目して独立に改造を行うことを可能にする方式について述べる．

### 2．プログラムスライシング

プログラムスライシング技術はプログラム内のある命令の実行に影響を与える可能性のあるすべての命令を抽出する技術であり，テスト，デバッグ，保守等，広範囲に応用されている　7)-12)．本章では，プログラムスライシング技術について概説する 13)-20).

#### （1）フローグラフ

分岐文の条件式の部分を分岐命令，ループ文の条件式の部分をループ命令と呼ぶ．また，代入文を代入命令，入力文を入力命令，出力文を出力命令と呼ぶこととする．

プログラムPのフローグラフとは，以下の条件を満たす有向グラフ G = (N, A, e) である．

(1) N はノードの集合で，ノードはプログラムP　内の命令からなる．

(2) A はアークの集合で，アーク (s, t) は，命令 s を実行した後，制御が直ちに命令 t へ移る可能性があることを示す．

(3) e はノードで，プログラムPを実行するときに最初に制御が移される命令である（e を開始ノードと呼ぶ）．

命令 s に対して，Def(s) は命令 s で定義される変数，Use(s) は命令 s で使用される変数の集合を表わすこととする．命令 s における変数 x の定義が命令 t に到達するとは，命令 s が変数 x を定義し，かつ，フローグラフ上で，s から t に至るパスが存在して，そのパス内には，他に変数 x を定義する命令が存在しない場合をいう（到達する定義の求め方については文献 21)を参照）．

（２）スタティックスライス

プログラムPのスライシング基準 C = (u, V) とは，次の２つ組をいう．

(1) u はプログラムP内の命令．

(2) Vはプログラム P 内の変数の部分集合．

プログラムPのスライシング基準 C = (u, V) に関するスタティックスライスとは，以下の条件を満たす，任意の実行可能なプログラム P' である 14).

(1) P' は，P から 0 個以上の命令を削除することにより得られたもの．

(2) 入力 x に対してプログラムPが停止するならば，プログラム P' も停止し，その時，プログラム P に入力 x を与えた時の実行系列（実行された命令の列）を EX，プログラム P' に入力 x を与えた時の実行系列を EX' とすると，

(3) 全ての変数 v ∈ V に対して，実行系列 EX' において命令 u を実行する直前における変数 v の値からなる列が，実行系列 EX において命令 u を実行する直前における変数 v の値からなる列に等しい．

スタティックスライスは，スライシング基準となる命令を起点とし，スライシング基準となる変数に関して，次に示す，命令の間のデータ依存関係と制御依存関係を逆向きにたどることによって取り出される命令の集合として求めることができる．

(a) データ依存関係 DD(s, t)

命令 s から命令 t に対してデータ依存関係 DD(s, t) があるとは，ある変数 w が存在して，命令 s における変数 w の定義が，変数 w を使用している命令 t に到達する場合をいう．

(b) 制御依存関係　CD(s, t)

命令 s から命令 t に対して制御依存関係 CD(s, t) があるとは，命令 s が分岐命令であり，命令 t がその分岐文内に直接含まれている場合，あるいは，命令 s がループ命令であり，命令 t がそのループ文内に直接含まれている場合をいう．命令 t が分岐文 S 内に直接含まれているというのは，分岐文 S 内の他の分岐文やループ文に命令 t が含まれていないことを意味する．

## 3．従来のソフトウェア改造方式

スタティックスライス（以下，単にスライスと略称）を利用した従来のソフトウェア改造方式について述べる．

### （1）改造による影響を波及させない改造方式

プログラムが A, B, C の３つの機能を実現している時，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 3)-5) 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．従って，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．改造の手順は次のようになる．

(1) まず，改造対象機能 A に関するスライス（SliceA）を求める．

(2) 次に，改造対象でない機能 B, C に関するスライス（SliceBC ）を求める．このスライスをコンプリメントと呼ぶ．

(3) ２つのスライスの差集合 SliceA - SliceBC を求める．この集合に含まれる命令を独立な命令と呼ぶ．

(4) コンプリメントを反転表示し，編集禁止にする．

(5) コンプリメントに影響を与えないような，独立な命令に対する編集のみを許可することにより，機能 A の改造をガイドする．

### （2）独立な命令に対する編集

コンプリメントに影響を与えないように改造するためには，改造後のプログラムにおける機能 B, C のスタティックスライスがもとのスライス SliceBC と変わらなければ良い．許可される独立な命令に対する編集には，(1) 独立な命令の削除，(2) 新しい変数の導入，(3) 独立な変数（独立な命令でのみ定義されている変数）に値を設定する代入命令の追加，(4) 制御フローを変えない分岐文の追加等がある．

### （3）プログラム WordCounter の改造

図１に示すプログラム WordCounter は，文字列からなるテキストファイルの行数 nl，語数 nw，文字数 nc をカウントして出力する．このプログラムでは，空白，タブ，

あるいは，改行で区切られた文字列を語として認識している．このプログラムの行数，文字数のカウント方法は変えないで，アルファベットからなる文字列を語として認識するように，このプログラムを改造する．

まず，語数に関するスライスを求める．カウントされた語数 nw は命令 17 で出力しているので，スライシング基準 (17, {nw}) に関するスライスを求める．図 2 において，左側に示した命令がスライシング基準 (17, {nw}) に関するスライスとなる．右側の雲形で囲まれた命令はスライスに含まれない部分（即ち，改造対象でない部分）を表わしている．スライスに含まれる命令の中，枠で囲まれた部分はコンプリメントに含まれる命令であり，その他の部分が独立な命令である．

アルファベットの後に空白が続いた時に語としてカウントする．このため，まず，直前に読み込んだ文字を記憶する新たな変数 ch を導入し，初期値を設定する（R1）．独立な分岐命令 10 を変更し（R2），独立な変数 ch に値を設定する代入命令を追加する（R3）．変数 InWord に値を設定していた独立な代入命令（命令 1, 11, 13 ）は削除する（図 3 ）．この一連の改造では，コンプリメントには影響を与えていないため，改造対象でない行数 nl，文字数 nc を計算している部分に対する再テストを行う必要がない．

```
1  InWord := NO;
2  nl := 0;
3  nw := 0;
4  nc := 0;
5  get(c);
6  while c ≠ EOF loop
7      nc := nc + 1;
8      if c = CR then
9          nl := nl + 1;
       end if;
10     if c = ' ' or c = CR or c = TAB then
11         InWord := NO;
       else
12         if InWord = NO then
13             InWord := YES;
14             nw := nw + 1;
           end if;
       end if;
15     get(c);
   end loop;
16 put(nl);
17 put(nw);
18 put(nc);
```

図 1　プログラム WordCounter

```
1       InWord := NO;

3       nw := 0;

5       get(c);
6       while c ≠ EOF loop




10              if c = ' ' or c = CR or c = TAB then
11                      InWord := NO;
                else
12                      if InWord = NO then
13                              InWord := YES;
14                              nw := nw + 1;
                        end if;
                end if;
15              get(c);
        end loop;

17      put(nw);
```

```
2       nl := 0;

4       nc := 0;


7       nc := nc + 1;
8       if c = CR then
9               nl := nl + 1;
        end if;
```

```
16      put(nl);

18      put(nc);
```

図2　語数 nw に関するスタティックスライス

```
R1      ch := NUL;

3       nw := 0;
```

```
5       get(c);
6       while c ≠ EOF loop
```

```
R2              if is_space(c) and is_alpha(ch) then
14                      nw := nw + 1;
                end if;
R3              ch := c;
```

```
15              get(c);
        end loop;
```

```
17      put(nw);
```

```
2       nl := 0;

4       nc := 0;


7       nc := nc + 1;
8       if c = CR then
9               nl := nl + 1;
        end if;
```

```
16      put(nl);

18      put(nc);
```

図 3　語数 nw に関する 改造

## ４．非改造部分から受ける影響

本章では，改造方法によっては，改造部分が非改造部分から影響を受けるため，正しく改造できないという問題について述べる．

### （１）課金プログラム Charge1 とその改造内容

図４に示す課金プログラム Charge1 は，各呼における通話開始時刻 ts, 通話終了時刻 te, 通話距離 d を入力して，通話料の総計 M と通話時間の総計 T を出力するプログラムである．TD(ts, te), TN(ts, te) は各々，各呼における昼間時間帯（8:00~19:00），夜間時間帯（19:00~8:00）における通話時間を返す関数，rs, rt は各々，昼間時間帯，夜間時間帯における課金率，R(d) は距離 d に応じた課金率を返す関数であるとする．通話料の総計 M に関して，次のような改造を行う場合を考える．

従来の夜間時間帯（19:00~8:00）を夜間時間帯（19:00~23:00）と深夜時間帯（23:00~8:00）の２つに分け，各々の課金率を ru, rv として，通話料の総計 M を出力するように，プログラムを改造する．各呼における深夜時間帯における通話時間を返す関数として，TL(ts, te) を用いることとする．課金プログラム Charge1 において，改造対象機能 M に関するスライスを求めると，図５に示すようになる．この改造対象機能 M に関するスライスに対して，次節の（２），（３）に示す２通りの方法で改造する場合（Charge2, Charge3）について考察する．

```
1   M := 0;
2   T := 0;
3   while get(ts, te, d) ≠ EOF loop
4       s := TD(ts, te);
5       t := TN(ts, te);
6       ms := s * R(d) * rs;
7       mt := t * R(d) * rt;
8       t := t + s;
9       T := T + t;
10      m := ms + mt;
11      M := M + m;
    end loop;
12  put(M);
13  put(T);
```

図４　課金プログラム Charge1

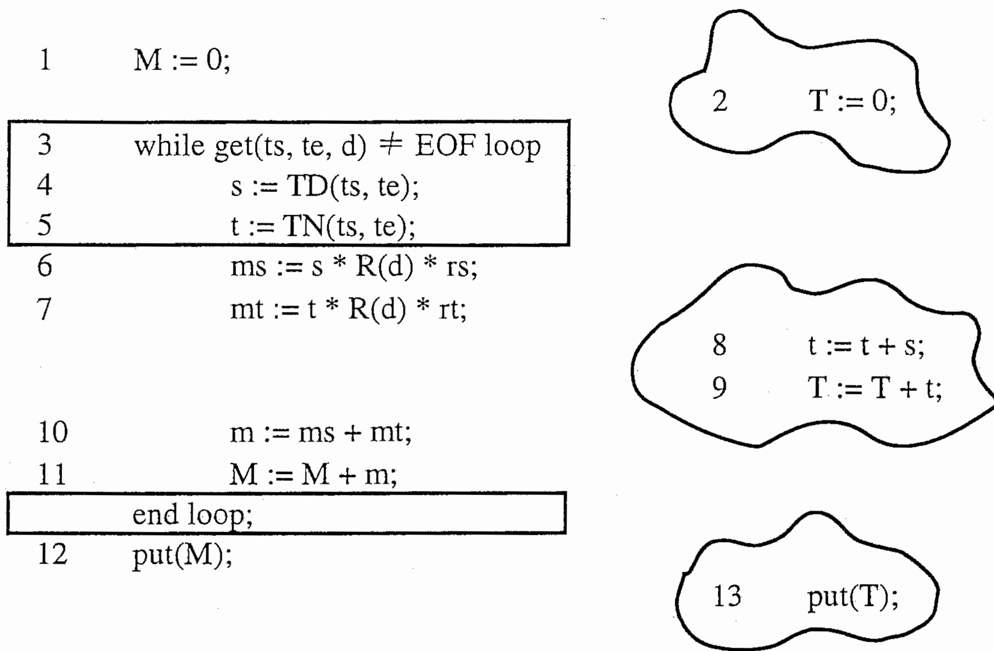



図５　通話料の総計Ｍのスタティックスライス

（２）改造プログラム Charge2

新たに設定した夜間時間帯（19:00~23:00）と深夜時間帯（23:00~8:00）の各々の通話料 mv, mu を求め，それらを昼間時間帯（8:00~19:00）の通話料 ms に加えるという方法により，改造を行う．改造結果を図６に示す．図中，Am は追加した命令，Rn は命令 n を変更（削除，追加）したことを表わす．命令 A1 で，深夜時間帯における通話時間 u を求め，命令 A2 で，新たに設定した夜間時間帯における通話時間 v を算出し，それらから命令 A3, A4 において，夜間時間帯と深夜時間帯の各々の通話料 mv, mu を求め，最後に，命令 R10 で，昼間時間帯も合わせた合計の通話料 m を求めている．旧仕様である夜間時間帯（19:00~8:00）の通話料 mt を算出していた命令７は不要なので削除している．

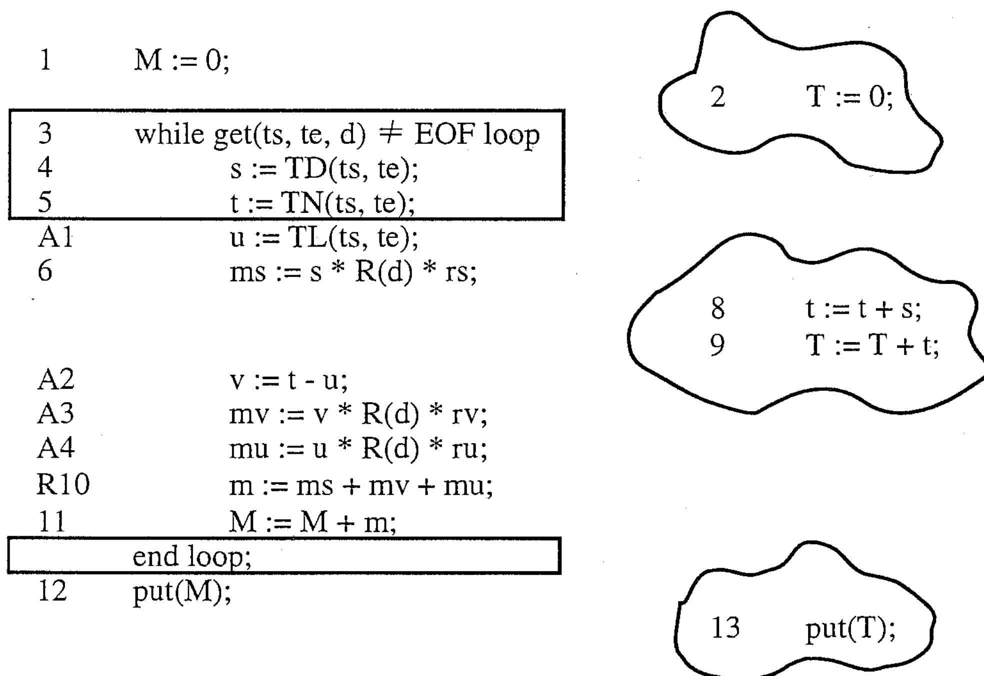



図６　課金プログラム Charge2

（3）改造プログラム Charge3

従来の通話料 m に対して，深夜時間帯（23:00~8:00）を新たに設けたことによる差額分だけを差し引くという方法により，改造を行う．改造結果を図7に示す．命令 A1 で，深夜時間帯における通話時間 u を求める処理は Charge2 と同じであるが，命令 A2 で，Charge1 で算出していた従来の通話料 m をそのまま用い，差額分だけを差し引いている．

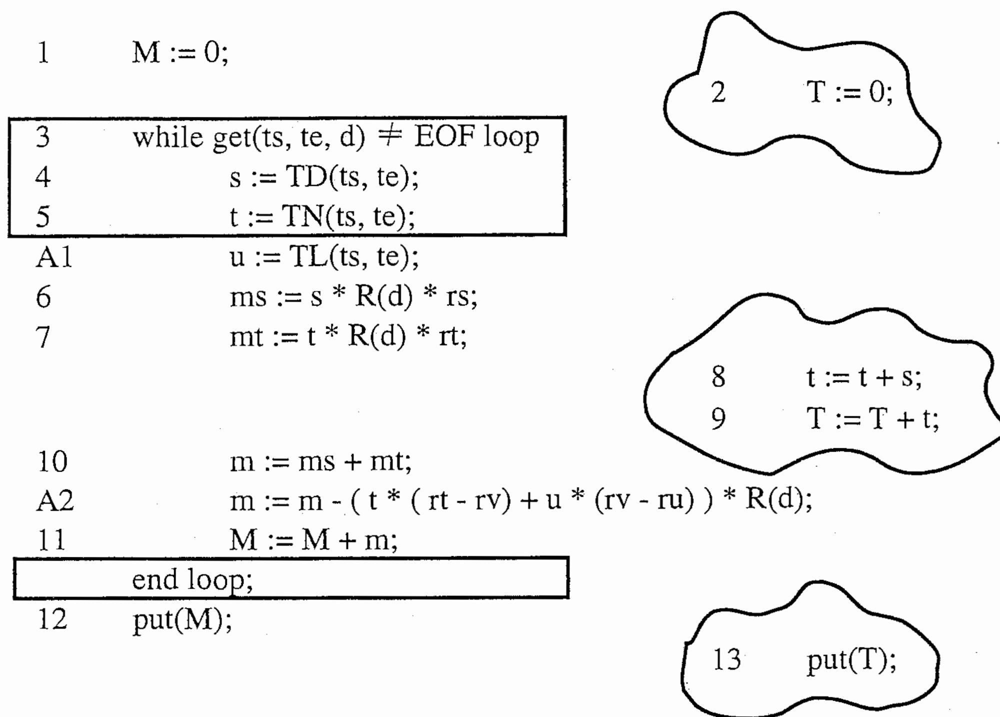



図7　課金プログラム Charge3

（4）改造部分が非改造部分から受ける影響

改造プログラム Charge2 において，命令 A2 では，命令 5 で算出した夜間時間帯（19:00~8:00）における通話時間 t を使用しているが，この変数 t は非改造部分内（命令 8）でも定義されている．そのために，この例の場合には，命令 A2 では，新たに設定した夜間時間帯（19:00~23:00）における通話時間 v の値を正しく求めることができない． Charge2 における改造は，非改造機能 T には影響を与えないが，非改造部分から改造部分が影響を受けるため，正しく改造できないという問題が起こっている．改造プログラム Charge3 においても，命令 A2 で，差額分を求めるために，命令 5 で算出した夜間時間帯（19:00~8:00）における通話時間 t を使用しており，このままでは，改造は正しく行われていない．

5．コード移動による影響の除去

改造部分が非改造部分から影響を受けたのは，改造部分で使用している変数に到達する定義が非改造部分内に存在したためである．改造プログラム Charge2 の例では，命令 A2 で変数 t を使用しているが，非改造部分内に存在する命令 8 における変数 t の定義が命令 A2 に到達している．この状況を定式化し，非改造部分から受ける影響を自動的に除去する方法について述べる．

（1）問題の定式化

プログラム P における改造対象機能（改造対象である値を出力する出力文と対応する変数からなるスライシング基準）を R1, R2, ..., Rm，非改造機能（改造対象でない値を出力する出力文と対応する変数からなるスライシング基準）を N1, N2, ..., Nn とする．以下の記号を導入する．

改造対象部分 $\mathrm{SliceR} = \bigcup_{i=1}^{m}$ ( Ri に関するスライス )

非改造対象部分 $\mathrm{SliceN} = \bigcup_{j=1}^{n}$ ( Nj に関するスライス )

独立部分 Ind = SliceR - SliceN

共有部分 Com = SliceR ∩ SliceN

非改造部分 NonRev = SliceN - SliceR

独立部分 Ind に含まれる命令が独立な命令であり，定義する代入命令がすべて独立な命令である変数，および，新しく導入する変数（異なる変数名をもつ新しい変数）

が独立な変数である．ここでは，独立部分に対する編集として，独立な変数 v へ値を設定する代入文 s を追加する場合について考える．この場合，ある変数 w ∈ Use(s) が存在して，文 s に到達する変数 w の定義が非改造部分 NonRev に存在すると，改造対象部分 SliceR に対して行った改造が，非改造部分から影響を受けることになる．言い替えれば，プログラマは改造対象部分 SliceR に着目して改造を行っているが，追加した文で使用している変数 w の値は，改造対象部分 SliceR 内で計算された値ではなく，プログラマの意図していない非改造部分 NonRev 内で計算された値が参照される可能性がある．従って，非改造部分 NonRev に存在する変数 w の定義が文 s に到達しないように，プログラムの意味を変えずにプログラムを変更することができればよい．ここでは，コード（文 s）を移動することにより，プログラムを変更する方法について考察する．

（2）コード移動の十分条件

独立な変数 v へ値を設定する代入命令 s を移動する位置を pos とする．位置 pos に関する要求条件は次のように記述できる．但し，ここでは，各命令は実行可能であるとする（コード移動によりゼロ割り等が発生する場合には，6章で述べるコード付加により影響の除去を行う）．

［要求条件］(Req 1) 命令 s を位置 pos に移動しても，改造対象部分プログラム SliceR の意味が変わらず（即ち，どんな入力を与えて実行しても，SliceR 内の，命令 s 以外の各命令の実行結果が変わらず），かつ，(Req 2) 非改造部分 NonRev 内に存在する変数 w ∈ Use(s) の定義が位置 pos に到達しない．

以下では，上記の要求条件を満足させるための十分条件について考察する．まず，要求条件 (Req 1) を満足する位置 pos について考える．改造対象部分プログラム SliceR に関して，以下の記号を導入する．

UseIns : 命令 s 以外の，変数 v を使用している命令の集合．

DefIns : 命令 s 以外の，変数 v を定義している命令の集合．

［定理 1］改造対象部分プログラム SliceR において，任意の命令 u ∈ UseIns, 命令 d ∈ DefIns に対して，以下に示す条件が成立することは，要求条件 (Req 1) が成立するための十分条件である．

(C1) 開始ノードから命令 u に至るパスで，変数 v を定義する命令を含まないものがあるならば，そのパスは位置 pos を含まない．

(C2) 命令 d における変数 v の定義が命令 u に到達するならば，その到達するパスは位置 pos を含まない．

(C3) 位置 pos は命令 s を支配する（即ち，フローグラフ上で，開始ノードから命令 s

に至るすべてのパスが位置 pos を通る）．

(C4) 条件 (C3) のもとでは，命令 s を位置 pos に移動すると，プログラムを実行した場合，もとの命令 s の位置 oldpos に制御が渡る前に必ず位置 pos に制御が渡っているが（位置 pos には複数回制御が渡っている場合もある），この時，位置 oldpos の直前の位置 pos で実行された移動後の命令 s の実行結果は，位置 oldpos で実行される移動前の命令 s の実行結果と変わらない．

(C5) 位置 pos から命令 s に至る，始点以外に pos を含まないすべてのパスが，命令 s 以外に変数 v を定義する命令を含まない．

（証明）要求条件 (Req 1) が成立するための十分条件は，任意の命令 u ∈ UseIns の実行時に使用する変数 v の値が命令 s を位置 pos に移動しても変わらないことである．最初に実行された命令 u について考えれば十分である．命令 u を実行した時，(1) 開始ノードから命令 u に至る実行されたパスの中に命令 d も命令 s も含まない場合，(2) 命令 d で設定された変数 v の値を使用した場合，(3) 命令 s で設定された変数 v の値を使用した場合とがある．(1) の場合には，(C1) より，命令 s を位置 pos に移動しても変数 v の値は変わらない．(2) の場合も，(C2) より，同様．(3) の場合には，(C3) より，命令 s は位置 pos で必ず実行されている．そして，(C4) より，直前の位置 pos で実行された命令 s の実行結果は変わらない．また，(C5) より，その命令 s で設定された変数 v の値が途中で再設定されることはないため，命令 s を位置 pos に移動しても変数 v の値は変わらない．□

条件 (C5) では，pos ---> s ---> s ----> s というパスにおいても，命令 s 以外に変数 v を定義する命令を含まないことを述べている．次に示すプログラムは，条件 (C5) を満足していない．< e1, s, e2, d, u, e1, s, e2, u > というパスに沿って実行する場合には，命令 s を位置 pos に移動すると，2 回目の命令 u の実行において，使用している変数 v の値を設定した命令が s から d へ変わってしまう．

```
pos;
while e1 loop
    s;               -- 変数 v を定義
    if e2 then
        d;           -- 変数 v を定義
    end if;
    u;               -- 変数 v を使用
end loop;
```

また，次の定理 2 が成立する．

［定理２］条件(C3)のもとで，条件(C4)が成立するための十分条件は，次の条件(C6)が成立することである．

(C6) 位置 pos から命令 s に至る，始点以外に pos を含まないすべてのパスが，任意の変数 w ∈ Use(s) に対して，変数 w を定義する命令を含まない．

（証明）位置 oldpos で実行された移動前の命令 s が命令 a で設定された変数 w の値を使用している場合には，(C6) より，命令 a から命令 s に至る実行されたパス内に位置 pos があるため，移動後の命令 s でも同じ命令 a で設定された変数 w の値を使用している．また，位置 oldpos で実行された移動前の命令 s が使用している変数 w の値が未定義の場合には，移動後の命令 s で使用する変数 w の値も未定義である．□

（３）コード移動アルゴリズム

条件 (C1), (C2), (C3), (C5), (C6)，および，(Req 2) を満たすような位置 pos を見つけ，その位置に命令 s を移動すればよい．ここでは，これらの条件を満足する位置 pos を決定するアルゴリズムについて述べる．

条件を満足する位置 pos が見つかった場合にはその位置（返された命令の直前が位置 pos を表わす）を返し，そうでない場合には Failure を返すコード移動アルゴリズムを図８に示す．まず，すべての変数 w ∈ Use(s) に対して，変数 w の定義が命令 s に到達するパスの集合 DefPaths(w) を予め求めておく（但し，各パスを表わす命令の列から，始点である変数 w を定義する命令，および，終点である命令 s を除外しておく）．改造対象部分プログラム SliceR において，命令 s を支配するノードの集合は，支配関係に関して全順序関係をもつ 21)．命令 a が命令 b を支配することを a < b で表わすこととし，命令 s を支配するノードの集合を dom[1] < dom[2] < ... < dom[n] （< s）とする．

domを始点とするパスを順に辿るには，分岐箇所において辿っていない分岐をスタックにプッシュダウンして，パスのトラバースを続け，既に辿ったノードに出会うか，あるいは，次に続くノードがない場合に，スタックをポップアップして次のパスを辿っていけばよい．パスを辿りながら，命令 u，命令 d，あるいは，命令 s に出会うかどうかを調べ，各命令に応じた処理を行う．最初に命令 u に出会わなければ，下記に示す定理３より，条件 (C1) および (C2) が満足されることがわかる．

コード移動アルゴリズムは，(1) 変数 w の定義が命令 s に到達するパスで dom[i] を含まないものがある，(2) 命令 s から dom[i] を通らないで命令 s に戻る，命令 d を含むパスがある，(3) dom[i] から，変数 v を定義する命令を通らずに命令 u に至るパスがある，あるいは，(4) dom[i] から dom[i] を通らずに命令 s に至るパスが命令 d を含む場合には， dom[i] を支配するノードはすべて，同様の状態になるため， Failure を返す．それ以外の場合には，次の支配ノード dom[i-1] について条件を満足するかどう

```
for 変数 w ∈ Use(s) loop
    変数 w の定義が命令 s に到達するパスの集合 DefPaths(w) を求める.
end loop;
i := n;
while i ≧ 1 loop
    dom := dom[i];                              -- 条件 (C3)
    if { ある変数 w ∈ Use(s) が存在して，DefPaths(w) 内のパスで dom を含まないもの
が
        ある } then
        return Failure;                         -- 条件 (C6)
    end if;
    if { 命令 s から dom を通らないで命令 s に戻る，命令 d を含むパスがある } then
        return Failure;                         -- 条件 (C5)
    end if;
    loop
        dom を始点とするパスを順に辿る.
        loop
            各パスにおいて，dom から到達するノードを順に調べる.
            if { 命令 u に出会う } then
                return Failure;                 -- 条件 (C7)
            elsif { 命令 d に出会う } then
                if { その命令 d から dom を通らずに命令 s に到達する } then
                    return Failure;             -- 条件 (C5)
                else
                    exit;
                end if;
            elsif { 命令 s に出会う } then
                exit;
            end if;
            exit when { 既に辿ったノードに出会う，あるいは，次に続くノードがない
};
        end loop;
        exit when { dom を始点とする，すべてのパスを調べた };
    end loop;
    if { 任意の変数 w ∈ Use(s) に対して，非改造部分 NonRev 内に存在する変数 w の
定義
        は dom に到達しない } then
        return dom;                             -- 条件 (Req 2)
    end if;
    i := i - 1;
end loop;
```

図 8　コード移動による影響除去アルゴリズム

かを調べていく.

［定理３］次の条件 (C7) が成立するならば，条件 (C1) かつ (C2) が成立する.

(C7) 位置 pos から命令 u に至るパスは，命令 s，あるいは，命令 s 以外の変数 v を定義する命令を含む.

［証明］命令 d における変数 v の定義が命令 u に到達し，その到達するパスが位置 pos を含むとすると，位置 pos から命令 u に至る，変数 v を定義する命令を含まないパスが存在するため，矛盾．開始ノードから命令 u に至るパスで，変数 v を定義する命令を含まないものがあり，そのパスが位置 pos を含むとすると，同様にして矛盾．
□

（４）改造プログラム Charge2 におけるコード移動

改造プログラム Charge2 では，命令 A2 で変数 t を使用していたが，非改造部分内に存在する命令 8 における変数 t の定義が命令 A2 に到達しているため，非改造部分から影響を受けている．非改造部分から受けるこの影響を除去するため，改造プログラム Charge2 に対して，コード移動アルゴリズムを適用してみる．

UseIns = { A3 }, DefIns = $\phi$, Use(A2) = { t, u } である．変数 t は命令 5 で定義しており，変数 t の定義が命令 A2 に到達するパスの集合は， DefPaths(t) = { < A1, 6 > } であり，変数 u は命令 A1 で定義しており，変数 u の定義が命令 A2 に到達するパスの集合は， DefPaths(u) = { < 6 > } である．命令 A2 を支配するノードは，命令 1 < 命令 3 < 命令 4 < 命令 5 < 命令 A1 < 命令 6 である．まず，命令 A2 を支配する命令 6 について考える．DefPaths(t), DefPaths(u) 内のパスは共に命令 6 を含む．命令 A2 から命令 6 を通らないで命令 A2 に戻るパスは存在しない．命令 6 を始点とするパスを辿ると，命令 A2 に出会う．調べるべきパスは他にないので（即ち，途中に分岐箇所はなかったので），次に，非改造部分 NonRev 内に存在する変数 t の定義（命令 8）を調べると，命令 6 には到達しないため，命令 6 が条件を満足する位置として返される．従って，命令 A2 を命令 6 の直前に移動すればよい（図 9）．

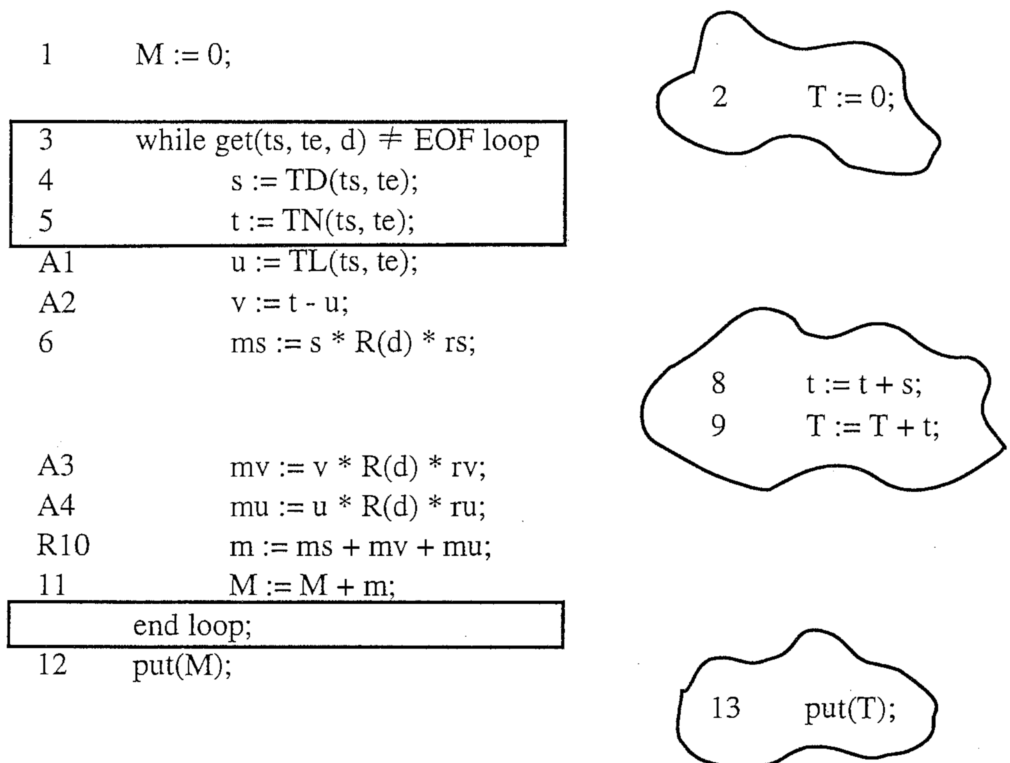



図 9　影響を除去した課金プログラム Charge2

## ６．コード付加による影響の除去

改造対象部分プログラム SliceR 内に独立な変数へ値を設定する代入文sが追加され，非改造部分 NonRev に存在する変数 w の定義が変数 w を使用している文 s に到達するとき，文 s をどんな位置に移動してもプログラムの意味を変えずに非改造部分から受ける影響を除去することはできない場合があることを示し，この場合にコードを付加することにより，プログラムの意味を変えずに非改造部分から受ける影響を除去する方法について述べる．

### （１）改造プログラム Charge3 におけるコード移動

改造プログラム Charge3 についても，コード移動アルゴリズムを適用し，非改造部分から受ける影響を除去することを考える．改造プログラム Charge3 では，命令 A2 で変数 t を使用していたが，非改造部分内に存在する命令 8 における変数 t の定義が命令 A2 に到達しているため，非改造部分から影響を受けている．

UseIns = { 11 }, DefIns = { 10 }, Use(A2) = { m, t, u, d } である．変数 m は命令 10 で定義しており，変数 m の定義が命令 A2 に到達するパスの集合は， DefPaths(m) = { < > } である．従って，命令 A2 を支配する命令 10 について考えた場合，DefPaths(m) 内のパスで命令　10を含まないものがあるため，Failure が返される．即ち，この改造プログラム Charge3 の場合には，命令 A2 を移動することにより，非改造部分から受ける影響を除去することはできない．そこで，必要最小限のコードの付加を行い，非改造部分から受ける影響の除去を行う．

### （２）コード付加アルゴリズム

コードは移動できないので，非改造部分から受ける影響を除去するため，非改造部分に存在する変数の定義が改造対象部分に到達しないように変数名の変更を行う．コード付加による影響除去アルゴリズムを図１０に示す．

文 s で変数 w を使用している場合，(1) 非改造部分 NonRev に存在する変数 w の定義は文 s に到達せず，かつ，(2) 改造対象部分プログラム SliceR において，文 s に到達する変数 w の定義は，プログラム P 全体においても到達するようにできればよい．そこで，改造対象部分プログラム SliceR において，文 s に到達する変数 w の定義の集合を Defs(s, w) とすると， Defs(s, w) 内の文 d （w := ....;）の直後に，代入文 "w_rev := w;" を挿入し，文 s が使用している変数の名前 w を w_rev に変更する．"_rev" は，変更した変数名を他の変数名と区別するための標識であり，区別できるものであれば何でもよい．

２つの文 s1, s2 ∈ S があり，どちらの文も同じ変数 w を使用しており，かつ，どちらの文も非改造部分から影響を受けている場合，改造対象部分プログラム SliceR にお

いて，各々の文 s1, s2 に到達する変数 w の定義の集合は同じであるとは限らない．しかし，次の定理4より，どちらの文の場合も，使用している変数の名前 w を同じ変数名 w_rev に変更してよい．

```
改造対象部分プログラム SliceR 内に追加された，独立な変数へ値を設定する代入文
の中，コード移動による影響除去ができない文の集合を S とする．
DeclVars := φ;
for 文 s ∈ S loop
    for 変数 w ∈ Use(s) loop
        if { 文 s に到達する変数 w の定義が非改造部分 NonRev に存在する } then
            改造対象部分プログラム SliceR において，文 s に到達する変数 w の定義
を全
            て求め，これを Defs(s, w) とする．
            DeclVars := DeclVars ∪ {w};
            for 文 d ∈ Defs(s, w) loop
                if { 文 d に対してコード付加はされていない } then
                    文 d の直後に，代入文 "w_rev := w;" を挿入する．
                end if;
            end loop;
            文 s が使用している変数 w を w_rev に変更する．
        end if;
    end loop;
end loop;
for 変数 w ∈ DeclVars  loop
    変数 w_rev の宣言を，変数 w の宣言の直後に挿入する．
end loop;
```

図１０　コード付加による影響除去アルゴリズム

［定理４］文 s1, s2 ∈ S に到達する変数 w の定義の集合を各々，Defs(s1, w), Defs(s2, w)　とする．図１０に示すコード付加による影響除去アルゴリズムを適用した場合，改造対象部分プログラム SliceR において，次のことが成立する．

(1) 文 d1 ∈ Defs(s1, w) の直後に挿入された変数 w_rev の定義は文 s1 に到達する．

(2) 文 d2 ∈ Defs(s2, w) - Defs(s1, w) の直後に挿入された変数 w_rev の定義は文 s1 に到達しない．

［証明］(1) 文 d1 ∈ Defs(s1, w) の直後に挿入された変数 w_rev の定義が文 s1 に到達しないとする．文 d1 における変数 w の定義は文 s1 に到達するため，その到達するパスは他の変数 w の定義を含まない．従って，そのパスには，文 d1 の直後に挿入された変数 w_rev の定義以外には変数 w_rev の定義を含まないため，矛盾．(2) 文 d2 の直後に挿入された変数 w_rev の定義が文 s1 に到達するパスが存在するとする．文 d2 における変数 w の定義は文 s1 に到達しないため，そのパス内には文 d2 以外の変数 w の定義 d で，文 s1 に到達するものがある．文 d の直後には変数 w_rev の定義が挿入されているため，矛盾．□

（３）改造プログラム Charge3 におけるコード付加

改造プログラム Charge3 では，非改造部分から受ける影響をコード移動により除去することはできなかったため，コード付加による影響除去アルゴリズムを適用する．S = { A2 }, Use(A2) = { m, t, u, d } である．文 A2 で使用している変数で，その変数の定義が非改造部分から到達するのは，変数 t である．Defs(A2, t) = {5} であるから，文 5 の直後に代入文 "t_rev := t;" を挿入し，文 A2 が使用している変数 t を t_rev に変更する（図１１）．最後に，変数 t_rev の宣言を，変数 t の宣言の直後に付け加える．

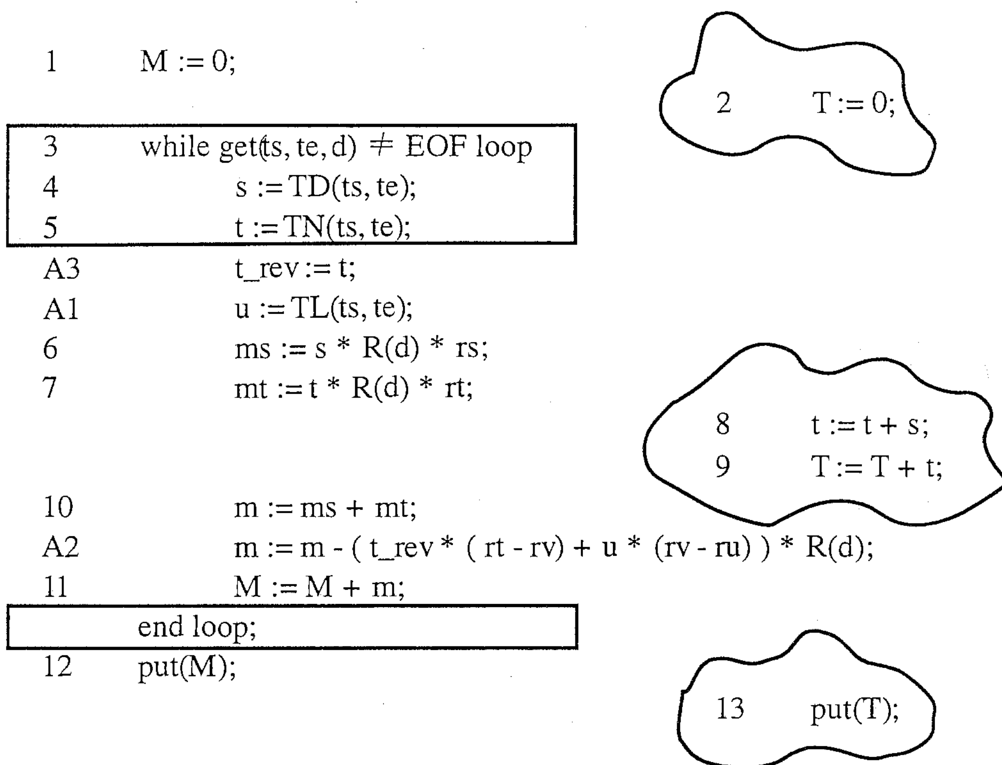



図１１ 影響を除去した課金プログラム Charge3

## 7．考察

### （1）スタティックスライスを用いた独立改造方式

プログラムのある機能を改造する場合には，その改造対象機能を実現している部分（改造対象部分プログラム SliceR）に着目して改造が行われる．本論文では，非改造対象部分 SliceN に影響を与えないように，改造対象部分プログラム SliceR に対して改造を行った後，その改造部分が非改造部分 NonRev から影響を受ける場合に，その影響を除去する方法について述べた．改造対象部分 SliceR はスタティックスライスであり，実行可能な部分プログラムである 14)．従って，改造対象部分SliceR だけを取り出して，独立に改造を行い，テストを行うことも可能である．そして，改造対象部分プログラム SliceR に対する改造／テスト作業が完了すれば，非改造部分から受ける影響が除去されているため，プログラム全体としての改造／テスト作業が完了したことになる．

Gallagher らの初期の研究 4), 5) では本論文で述べた，出力文とその出力文で出力する変数vの組をスライシング基準とするスタティックスライスを用いていたが，後に，変数 v の値を計算している部分をすべて取り出すため，変数 v の値を出力しているすべての文をスライシング基準とする decomposition slice を提案した 3)．また，Icuma ら 22) は，独立な変数に値を設定する代入文を追加する場合，その文を挿入すべき位置をプログラマが把握し易くするため，スライシング基準となる変数を定義しているすべての代入文をスライシング基準に加えることにより，Gallagher らの提案した decomposition slice を拡張している．本論文では，(1) スタティックスライスの方がこのような decomposition slice よりも，改造対象となる機能を実現している部分を正確に表現していること，および，(2) 挿入すべき位置はシステムが解析して，コード移動／コード付加等の適切な処理を行うことから，Weiser 14) の提案したスタティックスライスを用いている．

### （2）コード移動アルゴリズム

コード移動は，ループ不変文のループ外移動等のコンパイラの最適化処理でも用いられている．ループ内の文 s をループヘッダの前に移動できるための十分条件は次のようなものである 21)．文 s では変数 v を定義するものとする．

(L1) 任意の変数 w ∈ Use(s) に対して，文 s に到達する変数 w の定義はループ外にある．

(L2) 文 s はループのすべての出口を支配する．

(L3) ループ内で変数 v を定義する文は，文 s のみである．

(L4) ループ内での変数 v の使用に到達する変数 v の定義は，文 s のみである．

ループヘッダの前の位置を pos とすると，ループヘッダの性質から (C3) が，(L3)

から (C5) が，(L1) から (C6) が，また，(L2), (L4) から (C7) が導かれる．ループ不変文のループ外移動は，５（３）節で述べたコード移動アルゴリズムの特殊な場合となっている．

また，本論文では，独立な変数へ値を設定する代入文を追加する場合に，その文が非改造部分から受ける影響を除去する方法について述べた．制御の流れを変えない分岐文を追加し，その分岐文の分岐命令で使用している変数 w が非改造部分から影響を受ける場合には，その分岐文全体を移動すればよい．コード移動ができなかった場合には，代入文の場合と同様に，分岐命令で使用している変数 w に関してのみコード付加を行う．

（３）非改造部分を利用した改造

改造の内容によっては，非改造部分 NonRev 内で計算した変数の値を参照したい場合がある（例えば，図６に示した課金プログラム Charge2 において，改造対象部分プログラム SliceR 内では定義されていない変数 T の値等）．独立な変数 v へ値を設定する代入文 s を追加する場合，改造対象部分プログラム SliceR 内では定義されていないが，非改造部分 NonRev 内で定義されている変数 x を，文 s で使用する場合における，コード移動，および，コード付加の方法について考察する．

コード移動については，命令 s を位置 pos に移動しても，移動前の命令 s が実行された時に使用した変数 x の値を設定した命令が，移動後の命令 s が実行された時に使用した変数 x の値を設定した命令と同じであればよい．従って，変数 x に関しては，条件 (C6) の代わりに，次の条件 (C6x) を満足するようにコード移動を行いながら，文 s で使用している他の変数 w に関する影響の除去を行えばよい．

(C6x) プログラム P 全体において，位置 pos から命令 s に至る，始点以外に pos を含まないすべてのパスが，変数 x を定義する命令を含まない．

コード移動によって他の変数 w に関する影響の除去を行うことができなかった場合にはコード付加を行うことになるが，変数 x に関しては何もする必要はない．

## II部　分散処理システム障害解析支援方式の研究

### １．概要

システムのガイドに従ってバグを究明するアルゴリズミックデバッギングには，Shapiro1), PRESET2), GADT3), 4), PELAS5)-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 Slicing8) 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，変数値エラーに対して，Critical Sliceを用いた決定性のバグ究明方式を提案した9). Critical Sliceはエラーを引き起こす可能性のある文からなる集合であり，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，プログラムのテストにおいて観察される，出力異常エラー（変数値エラーを含む），出力漏れエラー，および，出力なし無限ループエラーの３つのエラーに対する Critical Slice に基づいたバグ究明方式を提案する．出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べ，そのような場合でも，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができることを示す．また，本バグ究明方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについても述べる．

### 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　バグの分類

プログラムの文に関するバグは，次の４つ，あるいは，これらの組み合わせからな

る．

(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) 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) 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呼ぶこととする．

## 2．2 Critical Sliceの定義

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変数の集合．

実行時点の間に，以下に示す4つの依存関係 Def, Ctl, CtlDef, OmsCondを定義する．

(1) 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．

(2) 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．

(3) 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．

(4) 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.

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

後で述べる出力異常エラーに対するバグ究明方式（3.2節参照）にも適用できるようにするため, 本論文では, 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, 関数 CEPを用いて, 次のように定義する.

CriticalEP(t, v) = CEP(t, {v}, t).

実行時点 i, e ( i ≦ e), 変数の集合 AffectUse に対して, 関数 CEP(i, AffectUse, e) は, 図 1 に示すように再帰的に定義される. 関数 CEPにおいて, Ctl(t) となる実行時点を CriticalEP(t, v) に含めているのは, 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においては, 1) 実行時点 tにおける制御フローは正しい（即ち, Ctl(t)内の実行時点における制御移行結果が正しい）が変数 vの値が誤っている場合と, 2) 実行時点 tにおける制御フロー自体が既に誤っている場合とがあるためである.

Critical 実行時点集合の要素を Critical 実行時点と呼ぶ. Critical 実行時点をノード, Critical 実行時点の間の Def, Ctl, CtlDef, OmsCond 関係をアークとすることにより定義されるグラフを Critical-Flow グラフと呼ぶ（図 3 参照）.

```
function CEP(i : 実行時点; AffectUse : 変数の集合; e : 実行時点) return 実行時点の集合
is
    X, EP : 実行時点の集合;
begin
    if i = e then
        X := Def(i, AffectUse) ∪ OmsCond(i, AffectUse) ∪ CtlDef(i, AffectUse) ∪ Ctl(i);
    else
        X := Def(i, AffectUse) ∪ OmsCond(i, AffectUse) ∪ CtlDef(i, AffectUse);
    end if;
    if X = φ then
        return ( φ ) ;
    end if;
    EP := X;
    for each x∈X loop
        EP := EP ∪CEP(x, Use(x), e);
    end loop;
    return EP;
end CEP;
```

図１　関数 CEP

(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, CriticalSlice(t, v) を以下のように定義する．

CriticalSlice(t, v) = { Ins(j) | j∈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 Slice10)-14) や Dynamic Slice15)-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．

## ２．３　Critical Sliceの性質

(1) 値誤りバグ潜在域

変数値エラーに対して，プログラム内の命令の集合Xで，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，その変数値エラーを引き起こすことがない場合，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呼ぶ．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．

（例）２つの値 x, y の min, max を求めるプログラム min_maxを図２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時の実行系列を図３に示す．図３において，jS は実行時点 jにおいて命令 Sが実行されたことを表す．この時，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）．CriticalEP(6, min) = { 1, 3, 4 } よ

り，CriticalSlice(6, min) = { Ins(1), Ins(3), Ins(4) } = { 1, 3, 4 } である．この例から分かるように，CriticalSlice内の命令 1, 3, 4 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ことが分かる．

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

図 2　　プログラム min_max

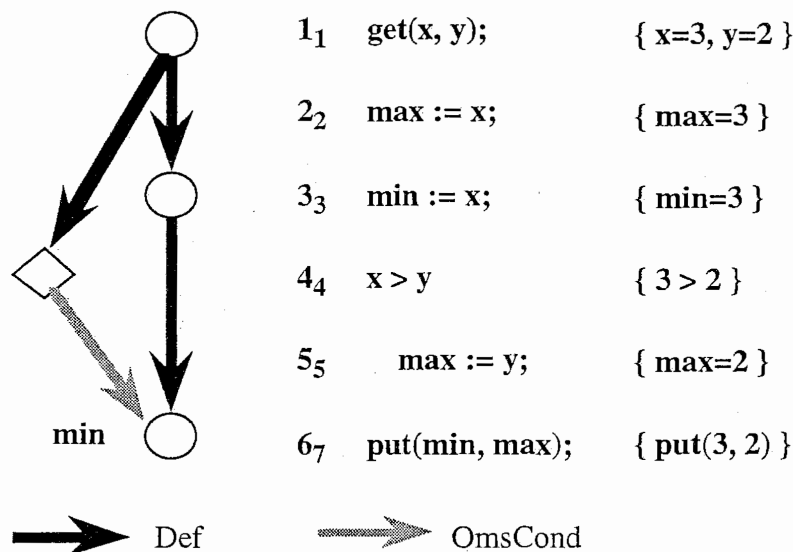



図 3　プログラム　**min_max** の実行系列

(2) 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 （$1 \leqq i \leqq t$）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) ＝ { 変数 x | $r < i \leqq s$ となる，ある r, s∈
CriticalEP(t, v) が存在して，x∈Use(s), r = Def(s, x) }.

FlowData(i) に含まれる変数をフローデータと呼ぶ．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．

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．実行時点 i ($1 \leqq i \leqq t$) の直前における制御フローが正しい（即ち，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，実行時点 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，実行時点 i 以後にバグが存在する．

従って，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（これを分割検証と呼ぶ）により，バグの存在範囲を限定することができる．

（例） 2 つの値 m, n の最大公約数を求めるプログラム gcd を図 4 に示す．プログラム gcdに入力 m=6, n=2 を与えて実行した時の実行系列を図 5 (a) に示す．値3が出力され，出力文の実行時における変数 xの値 3が誤っている（正しくは， x = 2）．出力文の実行時点 14における変数 xに関する Critical-Flowグラフを図 5 (a) に示す．バグは以下の手順で見つけることができる．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) 実行時点 8で分割する．まず，制御フローについて調べる．Ctl(8) = { 4 } であり，実行時点 4 におけるループ命令 "d≠0" の制御移行は正しい．そこで，次に，フローデータ c = 2, d = 3 の値の正誤を判定する（図 5 (a)）．ここで，フローデータでない変数 m, n, r については，エラーに関係しないため，それらの値を調べる必要はない．

変数 dの値 3が誤っている（正しい値は，d = 0）．

2) 実行時点 5で分割し，フローデータ c = 6, d = 2 の値の正誤を判定する（図 5 (b)）．

フローデータの値はすべて正しい．

3) 実行時点 6で分割し，フローデータ r = 3 の値の正誤を判定する（図 5 (c)）．

変数 rの値 3が誤っている（正しい値は，r = 0）．代入文 5で定義した変数 rの値 3は誤っており，代入文 5で使用した変数 c, dの値は正しいことから，代入文 5（"r := c / d;"）の文記述誤りバグが検出される．

```
1   get(m, n);
2   c := m;
3   d := n;
4   while d ≠ 0 loop
5       r := c / d;            (r := c mod d; が正しい)
6       c := d;
7       d := r;
    end loop;
8   x := c;
9   put(x);
```

図4　プログラム gcd

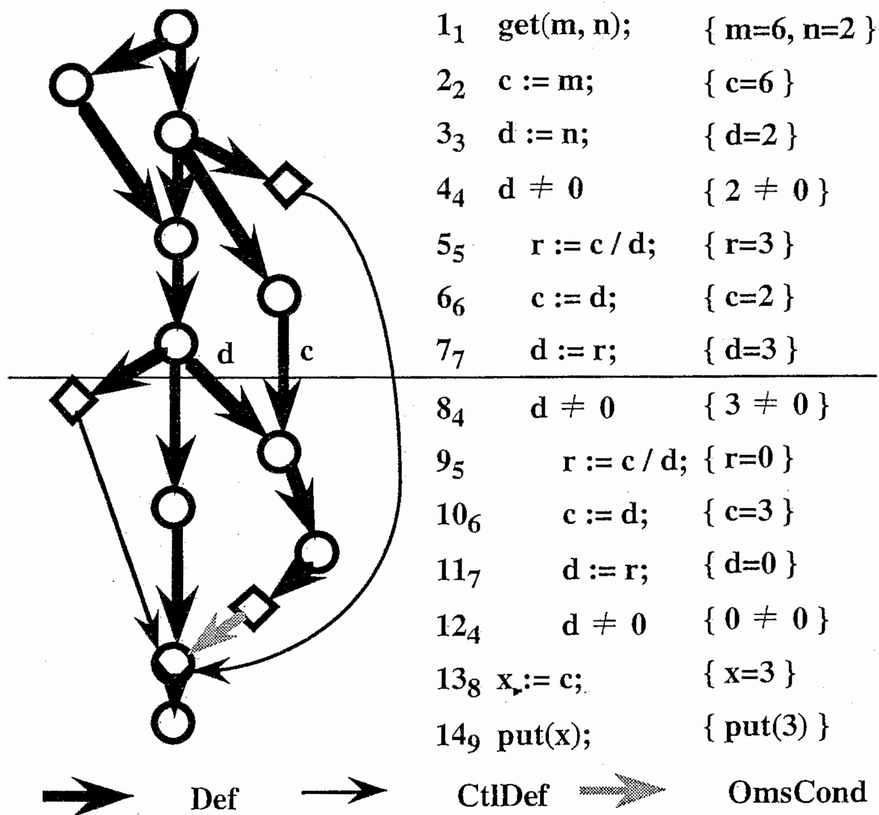



図5 (a)　プログラム　gcd のバグ究明

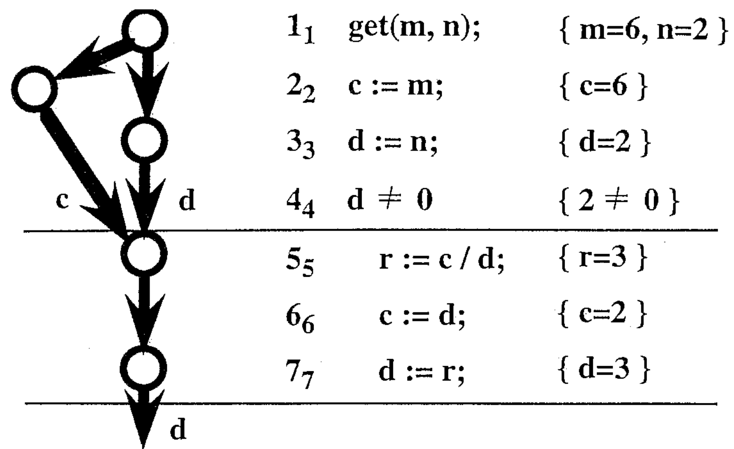



図 5 (b) プログラム **gcd** のバグ究明

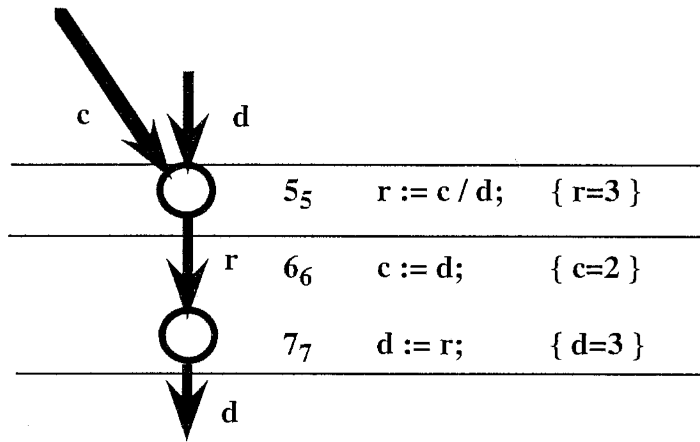



図 5 (c) プログラム **gcd** のバグ究明

## ３．バグ究明方式

### ３．１　出力に関するエラー

プログラムのテストにおいては，プログラムを実行した時の出力内容を観察することにより，誤り（以下，エラーと呼ぶ）を発見する．ここでは，プログラムのテストにおいて観察される，出力に関する次の３つのエラーに対するバグ究明方式について述べる．

(1) 出力異常エラー

出力内容の誤り，あるいは，余分な出力がある場合．この場合には，出力を実行した出力文を調べることにより，(a) 値の誤っている出力変数がある場合（変数値エラー）と，(b) 値の誤っている出力変数がない場合とに分かれる．同じ値を出力しながら，プログラムの実行が無限ループしている場合には，最初の異常な出力に関する出力異常エラーとして観察される．

(2) 出力漏れエラー

出力の漏れがある場合．

(3) 出力なし無限ループエラー

何の出力もなく実行が継続している場合．このエラーが観察された場合，3.3節で述べるバグ究明方式を適用していくと，プログラムが実際には無限ループしていないことが分かる場合もありうる．

### ３．２　出力異常エラーに対するバグ究明

出力異常エラーを引き起こした出力文を実行した実行時点を t とする．値の誤っている出力変数を v, ErrVar = {v} とする．出力文に制御が移行したこと自体が誤りであり，値の誤っている出力変数が存在しない場合には，ErrVar = $\phi$ とする．CEP(t, ErrVar, t) によって求められる Critical 実行時点に対して，分割検証を行い，バグを究明すればよい．

（例）与えられた値 xをもつ配列要素を捜し，その配列要素の添字の値を返すプログラム binary_search1を図６に示す．プログラム binary_search1に入力 x = 6, n = 2, A = (2, 6) を与えて実行した時の実行系列を図７(a) に示す．"Not found." が出力され，値 6をもつ配列要素が見つからない．この例では，値の誤っている変数の存在しない，出力異常エラーが発生している．

ErrVar = $\phi$ であるから，CEP(11, $\phi$, 11) によって求められる Critical実行時点を分割検

証する．実行時点 11における，Critical-Flowグラフを図 7 (a) に示す．バグは以下の手順で見つけることができる．

1) 実行時点 9で分割し，フローデータ i, j, x, k, A[1] の値の正誤を判定する（図 7 (a)）．

変数 iの値 1，および，変数 jの値 0が誤っている（この時点における正しい値は，i = 2，j = 2 である）．変数 iに着目して（詳細は割愛するが，この選択が最適である），バグを究明してみる．

2) 実行時点 7で分割し，フローデータ i, x, k の値の正誤を判定する（図 7 (b)）．

フローデータの値はすべて正しい．

3) 実行時点 8で分割し，実行時点 7における分岐命令の制御移行結果の正誤を判定する（図 7 (c)）．

分岐命令の制御移行結果が誤っており，その分岐命令で使用した変数の値はすべて正しいことから，分岐命令 7（"x < A[k]"）の文記述誤りバグである．

```
1   get(x, n, A);
2   i := 1;
3   j := n;
4   k := 1;
5   while  A[k] ≠ x and i ≦ j  loop
6         k := (i + j) / 2;
7         if  x < A[k]  then          (x > A[k] が正しい)
8               i := k + 1;
            else
9               j := k - 1;
          end if;
     end loop;
10  if  A[k] = x  then
11        put(k);
      else
12        put("Not found.");
    end if;
```

図 6　プログラム binary_search1

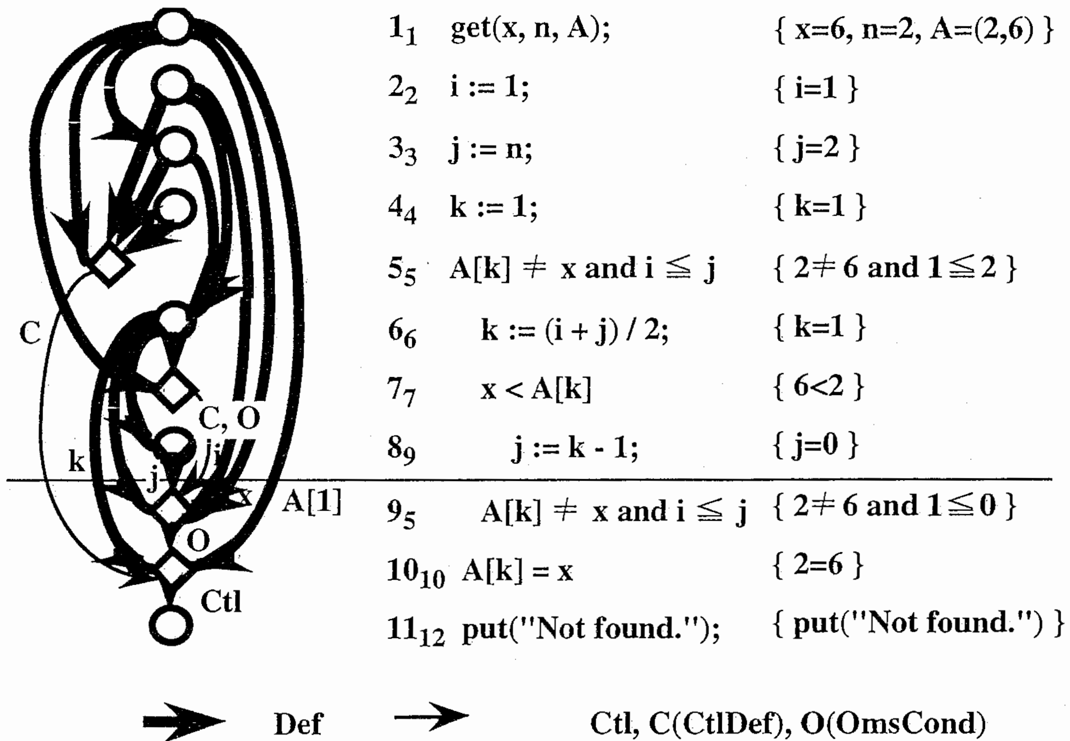



図 7 (a) プログラム binary_search1 のバグ究明

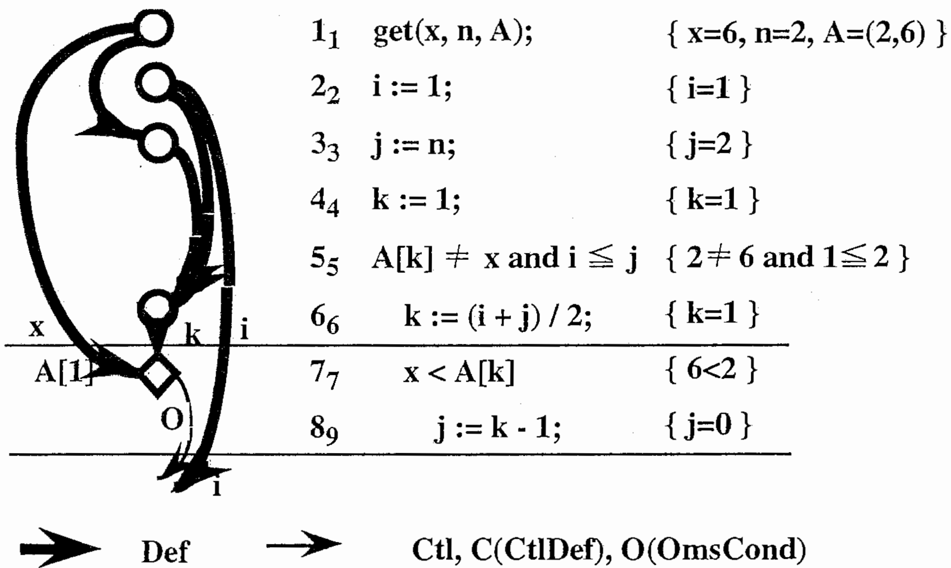
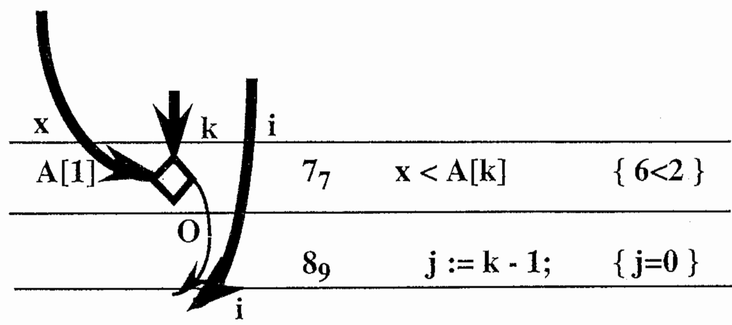



図 7 (b) プログラム binary_search1 のバグ究明




Def Ctl, C(CtlDef), O(OmsCond)

図 7 (c) プログラム **binary_search1** のバグ究明

## ３．３　出力なし無限ループエラーに対するバグ究明

出力なし無限ループエラーが観察される場合には，プログラムの実行を強制中断し，強制中断した時点 iを分割点と考え，制御フローの検証（Ctl(i)に含まれる実行時点における制御移行結果の正誤を順に判定すること）を行う．

分割点 i を含むループ文については，そのループ文内を繰り返し実行すべき回数をnとすると，n + 1 回以上，繰り返されている場合には，n + 1 回目に実行されたループ命令の制御移行結果の誤りである．そのループ命令を実行した実行時点を t，そのループ命令で使用した変数で値の誤っているものを v とすると，CEP(t, {v}, t) によって求められる Critical実行時点に対して分割検証を行い，バグを究明すればよい．ループ命令で使用した変数で値の誤っているものが存在しなければ，そのループ命令の文記述誤りバグである．

ループ文内の繰り返し回数が n以下である場合には，最後に実行されたループ文の繰り返し部分の実行系列に対して，制御フローの検証を続ける．この場合には，このループ文の中に入れ子になっている別のループ文が無限ループしている可能性がある．また，制御フローの検証の結果，Ctl(i) に含まれるすべての実行時点における制御移行結果が正しいと判定された場合には，プログラムは実際には無限ループしていなかったことになる．

（例）binary_search1 とは別の箇所にバグを含むプログラム binary_search2 を図８に示す．プログラム binary_search2 に入力 x = 6, n = 2, A = (2, 6) を与えて実行した時の実行系列を図９(a) に示す．プログラム binary_search2は，出力なしで，無限ループを実行する．

実行を強制中断し，実行系列の検証を行う．ループ文 5は，ループ文内を２回だけ実行して，ループから出なければならない．従って，３回目に実行されたループ命令の制御移行結果の誤りである．そのループ命令で使用した変数で値の誤っているものは，変数 k=1 と変数 i=1 である（正しくは，k=2, i=2）．ここでは，変数 kに着目して（この選択が最適である），バグを究明する．実行時点 13における変数 kに関する Critical-Flowグラフを図９(a) に示す．バグは以下の手順で見つけることができる．

1) 実行時点 10で分割し，フローデータ i, j の値の正誤を判定する（図９(a)）．
   変数 iの値 1が誤っている（正しい値は 2）．
2) 実行時点 8で分割し，フローデータ k の値の正誤を判定する（図９(b)）．
   フローデータ k の値 1は正しい．
3) 実行時点 9で分割し，フローデータ i の値の正誤を判定する（図９(c)）．
   代入文 8で定義した変数 iの値 1は誤っており，代入文 8で使用した変数 kの値は正しいことから，代入文 8（"i := k;"）の文記述誤りバグである．

```
1   get(x, n, A);
2   i := 1;
3   j := n;
4   k := 1;
5   while  A[k] ≠ x and i ≦ j  loop
6       k := (i + j) / 2;
7       if  x > A[k]  then
8           i := k;              ( i := k + 1; が正しい)
          else
9           j := k - 1;
        end if;
    end loop;
10  if  A[k] = x  then
11      put(k);
      else
12      put("Not found.");
    end if;
```

図 8　プログラム binary_search2

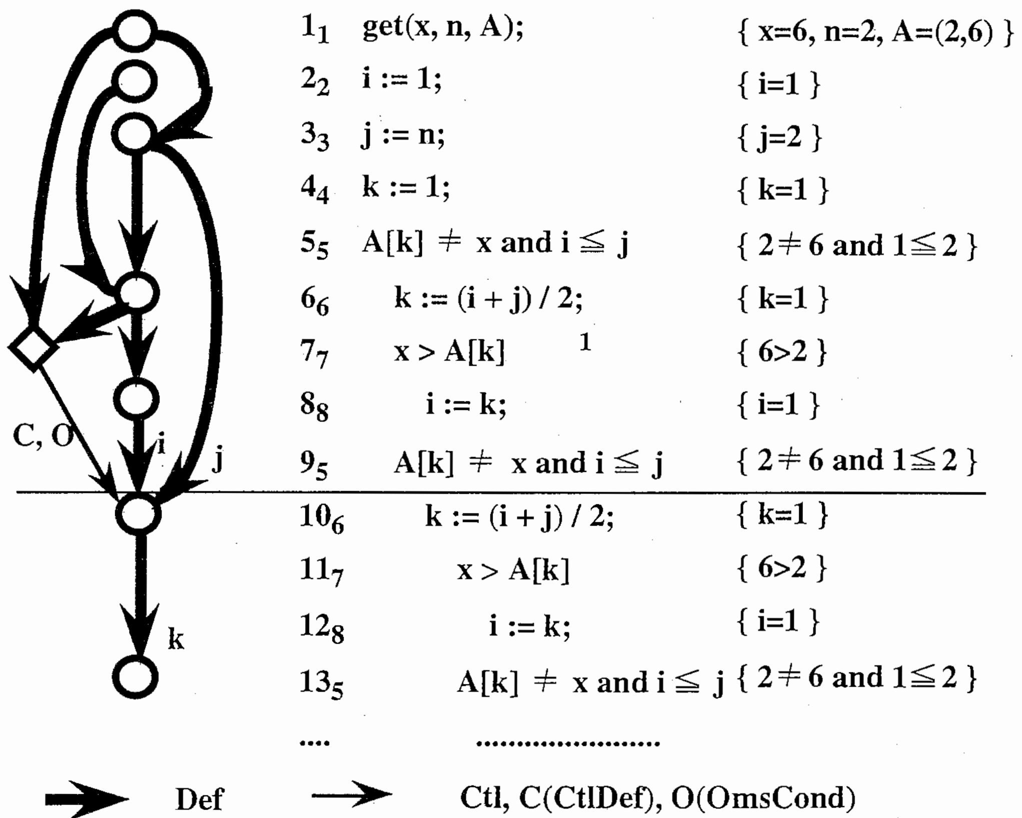



図 9 (a)　プログラム　**binary_search2** のバグ究明

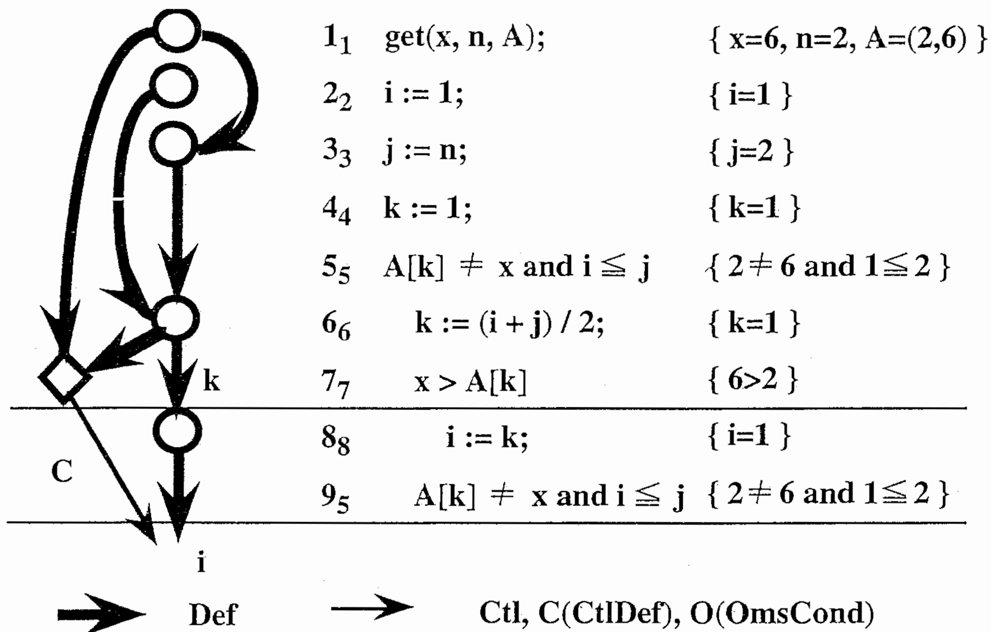



図９(b)　プログラム　**binary_search2** のバグ究明

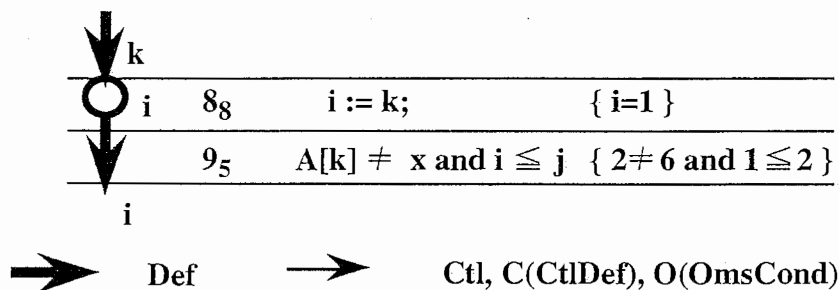



図９(c)　プログラム　**binary_search2** のバグ究明

３．４　エラー誤認の回避

ここでは，まず，出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べる．次に，そのような場合でも，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができ，出力漏れエラーを含むすべてのエラーに対して，バグを検出できることを示す．

(1) エラーの誤認

図１０(a) に示した正しいプログラム prog_out1 に入力 n＝2 を与えて実行させると，出力は，"AB" となる．一方，図１０(b) に示したバグを含んだプログラム prog_out2 に入力 n = 2 を与えて実行させると，出力は，"BB" となる．プログラム prog_out2 の実行では，最初の Bの出力の直前に，Aの出力が漏れていて，かつ，最初の Bの出力の直後に，余分な Bの出力がある．

しかし，プログラマは，プログラム prog_out2 の誤った実行結果を見て，２つの B-の出力の中，２番目の Bの出力は正しいが，最初の Bの出力が誤っていると誤認する

可能性がある．即ち，実行時点 4 において，値の誤っている出力変数が存在しない出力異常エラーが発生していると認識した場合には，CEP(4, $\phi$, 4) によって求められるCritical実行時点に対して，分割検証を行うことになる（図１１）．まず，実行時点 3-で分割すると，ブローデータ nの値 2は正しい．そこで，次に，実行時点4で分割すると，実行時点 3における分岐命令（n≧2）の制御移行結果も正しい．従って，出力異常エラーではなかったことが分かる．

```
1  get(n);
2  if n≧1 then
3      put('A');
   end if;
4  if n≧2 then
5      put('B');
   end if;
6  if n≧3 then
7      put('C');
   end if;
```

(a)　正しいプログラム prog_out1

```
1  get(n);
2  if n≦1 then          (n≧1 が正しい)
3      put('A');
   end if;
4  if n≧2 then
5      put('B');
   end if;
6  if n≦3 then          (n≧3 が正しい)
7      put('B');        (put('C'); が正しい)
   end if;
```

(b)　バグを含んだプログラム prog_out2

```
CP : integer := 0;
procedure put(c: in character) is
begin
   CP := CP + 1;
   OUT[CP] := c;
end;
```

(c)　手続き put

図１０　エラー誤認の回避

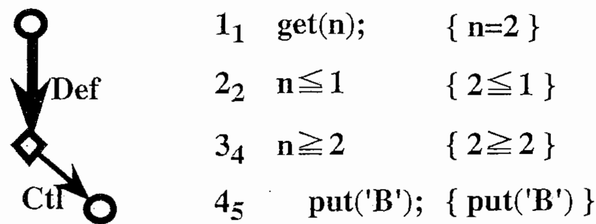



図１１　　プログラム　**prog_out2** の **Critical-Flow**グラフ

(2) 出力依存関係の導入

プログラマがエラーを誤認する可能性がある場合には，出力文の間の依存関係を導入することにより，プログラマによるエラーの誤認を回避し，バグを的確に究明することができる．言い換えれば，プログラマがプログラム prog_out2の実行結果を見て，最初の Bの出力が誤っていると認識しても（これは自然な認識であるが），バグを見つけることができる．

出力文を実行すると，配列　OUTに出力された値が順に格納されるものとみなす．即ち，手続き put では，図１０(c) に示す処理を行うと考える．従って，プログラム prog_out2では，図１２(a)に示す処理を実行するとみなす．そして，プログラマが認識した，最初の Bの出力に関する出力異常エラーは，出力文を実行した時点における配列要素 OUT[CP] の値に関する変数値エラー，即ち，変数 CP と変数 c に関する変数値エラーとして扱う．出力異常エラーに対するバグ究明方式を適用し，CEP(6, {CP}, 6) によって求められる　Critical実行時点の分割検証を行うことにより，以下の手順でバグを見つけることができる．

1) 実行時点 4で分割し，フローデータ n（= 2），CP（= 0）の値の正誤を判定する（図１２(a)）．
   この時点では，既に出力が行われていなければならないため，CP の値 0が誤っている．
2) 実行時点 3で分割し，フローデータ n（=2），CP（=0）の値の正誤を判定する（図１２(b)）．
   フローデータ n, CP の値は正しい．
3) 実行時点 3における分岐命令（n≦1）の制御移行結果の正誤を判定する．
   分岐命令の制御移行結果は誤っており，その分岐命令で使用した変数の値はすべて正しいことから，分岐命令（n≦1）の文記述誤りバグである．

出力依存関係を導入した場合には，出力漏れは次の出力の出力異常エラーとして認識される．次の出力が存在しない場合には，プログラムの実行終了時点における変数 CP の値の変数値エラーとして認識すればよい．

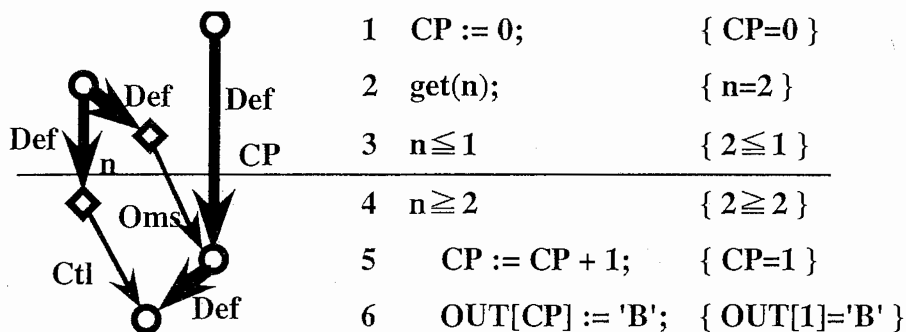



図１２(a) プログラム **prog_out2** のバグ究明

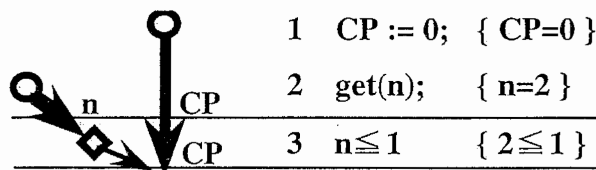



図１２(b) プログラム **prog_out2** のバグ究明

４．実現方式

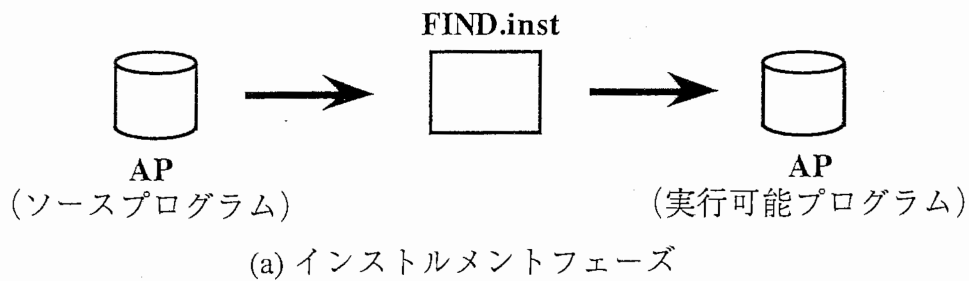
手続き型言語に対して，本論文で提案するアルゴリズミックデバッギングを実現するための方式について述べる．ここでは，SUN/UNIX ワークステーション上に試作中のデバッギングシステム FIND（Fault-locating INtelligent Debugger）を例にとり，システムの構成，ユーザインタフェース，および，処理内容について，具体的に記述する．FINDは C 言語で記述されたプログラムの出力異常エラーに対するバグを検出する．

４．１　処理形態

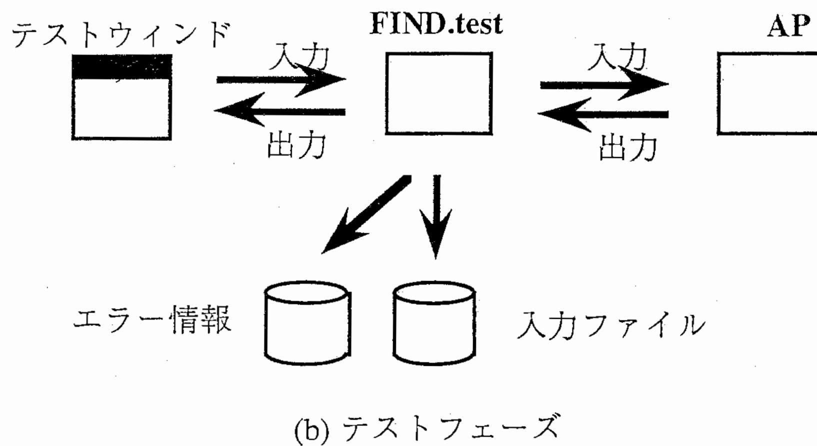
FINDにおけるバグ究明処理は，次の３つのフェーズから構成される．

1) インストルメントフェーズ -- デバッグ対象プログラム（以下，AP とする）に，次のテストフェーズ，および，デバッグフェーズにおいて AP の実行を制御するために必要なソースコードを埋め込んだ後，コンパイル，リンクを行い，実行可能なプログラムを生成する．
2) テストフェーズ -- AP を実行してテストする．
3) デバッグフェーズ -- テストフェーズで発見された出力異常エラーに対して，バグを検出する．

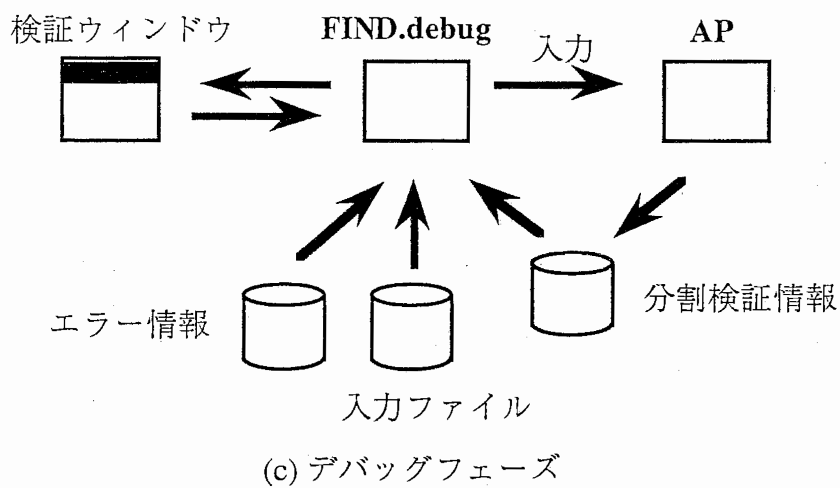
システム構成を図１３に示す．




(a) インストルメントフェーズ




(b) テストフェーズ




(c) デバッグフェーズ

図１３　システム構成

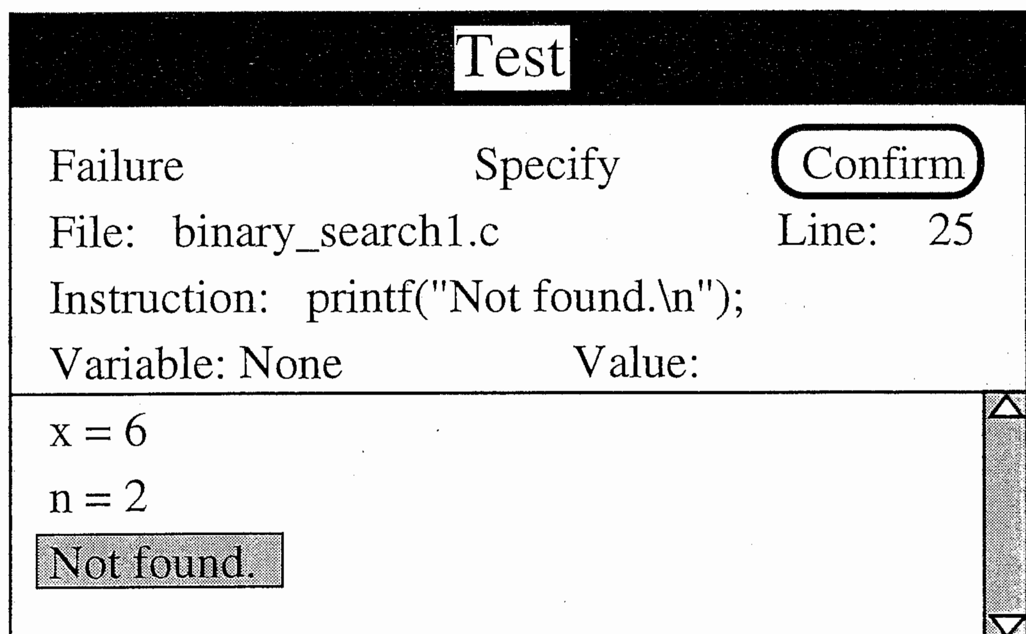
## ４．２　ユーザインタフェース

(1) テストフェーズ

テストウィンドウに AP への入力および AP からの出力が表示される．出力内容の中の誤っている部分をマウスで選択することにより，ユーザがエラーを指摘する．システムが選択された部分に対応する出力文，および，誤った出力を行った変数名，値を表示するので，ユーザはエラー内容を確認する．

(2) デバッグフェーズ

システムが提示した実行系列内の分割点に対して，ユーザが検証ウィンドウ上で制御フローやフローデータの値の判定を行う．これを繰り返すと，最後に，エラーを引き起こした原因となるバグをシステムがユーザに提示する．ユーザは，検証ウィンドウで検証中の命令に対応する箇所を，ソースプログラムや実行系列を表示したウィンドウ上で参照することができる．テストウィンドウおよび検証ウィンドウの表示例を図１４に示す．




(a) テストウィンドウ

Verification

Step: 1 Confirm

Control Flow

1 Loop ( 1) OK NG

A[k] != x && i <= j

Division Point: A[k] != x && i <= j

Flow Data

1 i = 0 OK NG

2 j = -1 OK NG

3 k = 0 OK NG

(b) 検証ウィンドウ

図１４　ユーザインタフェースの表示例

## ４．３　テストフェーズの処理

1) FIND が fork(), execve() の実行により AP を起動する．

2) ユーザから入力された入力データを パイプを通してAP に送るとともに，入力データファイルに記録する．

3) AP からの出力はパイプを通して受取り，テストウィンドウに表示する．この時，出力内容の表示位置と，その出力を行った出力文の実行時点，および，値を出力した変数との対応を記録しておく．

4) ユーザが出力内容の中の誤っている部分をマウスでクリックした時，出力内容の表示位置との対応関係を基にして，出力異常エラーが発生した実行時点，および，値の誤っている変数をエラー情報として記録し，デバッグフェーズに引き継ぐ．

## ４．４　デバッグフェーズの処理

1) FIND が fork(), execve() の実行により AP を再起動する．

2) 入力データファイルに記録された入力データを パイプを通して AP に送り，AP を自動的に再実行する．

3) エラー情報を基に，AP を再実行しながら，実行系列内の分割点を決定し，分割点における制御フローやフローデータの値の判定に必要な情報を獲得する（4.5節参照）．

4) 分割点における制御フローとフローデータの値を検証ウィンドウに表示し，ユーザにそれらの正誤を判定させる．

5) 以上を繰り返しながら，分割検証を進め，最後に，バグを検出する．

### 4．5　分割検証に必要な情報の獲得

分割検証を行うためには，Critical 実行時点集合や Critical Slice を必ずしも求める必要はない．必要となるのは，分割点，および，分割点における制御フローとフローデータ（これらを分割検証情報と呼ぶ）である．以下では，AP を実行させながら，これらの分割検証情報を獲得する方法について述べる．

(1) AP 実行時に記録する情報

分岐命令あるいはループ命令を実行する場合には，その実行時点 p をスタック CurrentCtl にプッシュダウンし，分岐文，ループ文の実行終了時には，実行時点をポップアップする．代入命令を実行し，変数 w を定義する場合には，その実行時点 d を LastDef(w) に設定し，実行時点 d における CurrentCtl を LastDefCtl(w) に設定する．

また，分岐命令あるいはループ命令を実行する実行時点 p において，w ∈ OmsVars(p) となる変数 w に対して，実行時点 p を OmsCondCandidates(w) に追加する．OmsVars(p) は，実行時点 p における制御移行が変わると，定義される可能性のある変数の集合である．これは，分岐文の then 節，else 節，あるいは，ループ文内で定義される可能性のある変数の集合であり，分岐命令あるいはループ命令毎に，インストルメントフェーズで求めておく．代入命令を実行し，変数 w を定義する場合には，OmsCondCandidates(w) を空集合にする．

(2) 分割点の決定

ユーザが分割点で行う判定を容易にするため，検証対象範囲にある実行時点の数を2分する実行時点の近傍で，分岐命令やループ命令のネストレベルがなるべく小さいところを分割点として選択する．

(3) 分割点における制御フロー

分割点 i に対して，Ctl(i) に含まれる各実行時点 p で実行された分岐命令あるいはループ命令の実行結果（True，False），および，それらの命令で使用された変数の集合 Use(p) と，それらの変数の値を求める必要がある．このため，検証対象範囲内にある，分岐命令あるいはループ命令を実行する実行時点pでは，スタック CurrentCtl に，実行時点 p の他に，命令 Ins(p)，その実行結果，変数の集合 Use(p)，および，それらの変数の値をプッシュダウンする． Ctl(i) に関する情報は，分割点 i における CurrentCtl から求めることができる．

(4) 分割点におけるフローデータ

分割点 i におけるフローデータの変数名とその値を求める必要がある．まず，$i \leqq j$ となる各実行時点 j において，各変数 w ∈ Use(j) に対して，次のように，Def(j, w),

CtlDef(j, w), OmsCond(j, w) を求める．

Def(j, w) := LastDef(w);

CtlDef(j, w) := LastDefCtl(w) - CurrentCtl;

OmsCond(j, w) := OmsCondCandidates(w) - CurrentCtl;

Def(j, w) $<$ i ならば，変数 w が分割点 i におけるフローデータの候補となるため，変数 w とその値を，実行時点 j に付随するリンク情報 LinkInf(j) に記録する．次に，各実行時点 k $\in$ Def(j, w) $\cup$ CtlDef(j, w) $\cup$ OmsCond(j, w) に対して，i $\leqq$ k ならば，実行時点 k に付随するリンク情報 LinkInf(k) を，実行時点 j に付随するリンク情報 LinkInf(j) に追加する．実行時点 t において，変数 v の値が誤っているという変数値エラーに対するバグを究明する場合には，分割点 i におけるフローデータに関する情報は，実行時点 t の変数 v に関するリンク情報から求めることができる．

おわりに

「I部　副作用のないソフトウェア改造方式の研究」では，ソフトウェアの改造を行う場合，改造対象機能を実現している部分だけに着目して，改造作業を進め，テスト作業も改造部分についてだけ行えば，プログラム全体に対してテストを行ったのと同じ効果を持つソフトウェア独立改造方式について述べた．改造部分が非改造部分から影響を受ける場合には，コードを移動，あるいは，コードを付加することにより，その影響を除去した．プログラムのバグの修正や部分的な改造等の保守への応用が考えられる．

本ソフトウェア改造方式におけるソフトウェア改造プロセスを９５年度委託研究より得られたソフトウェア計量化技術を活用して評価したところ，部分プログラムの抽出，改造，テスト，部分プログラムの影響解析，部分プログラムの影響除去，部分プログラムの統合というプロセスでは，改造，テスト以外は，システムによる完全自動化が図られており，また，改造もシステムが非改造部分へ影響を与えないようにガイドすることから，適切なプロセスであることが確認できた．

今後の課題としては，以下の項目が考えられる．

1) 出力文以外の任意の命令を対象とした改造方式
2) 改造に伴う影響を複数コードの同時移動により除去する方式
3) システムcall 等を含む実用プログラムのスライシング方法
4) シングルプロセスを対象とした改造方式の分散処理システムへの拡張方法
5) STR記述による通信処理プログラムの改造への適用方法等．

「II 部　分散処理システム障害解析支援方式の研究」では，出力異常エラー，出力漏れエラー，および，出力なし無限ループエラーの３つのエラーに対するアルゴリズミックデバッギング方式について提案し，本方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについて述べた．

出力異常エラーの発生している実行時点をユーザが直接，意識しなくてもすむようにするため，テストウィンドウ上でユーザがエラーを指摘する方式を採用した．また，テストフェーズですべてのテストをまとめて行い，エラーを発見する毎に，そのエラーに対するバグを究明するために必要となる分割検証情報を，バックグラウンドジョブとして別のプロセスで予め収集しておくと，デバッグフェーズでは，ユーザが直ちに分割検証を開始することができる．

今後の課題としては，以下の項目が考えられる．

1) 分割点における処理の文脈を分かり易く表示する方式
2) 値誤りバグ潜在域を最適化を実現する方式
3) １度正しく実行された命令にはバグを含む可能性が低いという仮定の下に，分割

検証ステップを効率化する方式

4) FIND システムのバグ究明効率，操作性に関する評価

5)　シングルプロセスを対象としたバグ究明方式の分散処理システムへの拡張方法等.

## 参考文献

「Ｉ部　副作用のないソフトウェア改造方式の研究」

1) Korel, B. and Sherlund, B.: Modification Oriented Software Testing, Proceedings of The Eighth International Conference on Testing Computer Software, pp. 143-152 (June 1991).

2) Leung, H. K. N. and White, L.: Insights into Regression Testing, Proceedings of Conference on Software Maintenance-1989, pp. 60-69 (October 1989).

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

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

5) 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).

6) Oman, P.: Maintenance Tools," IEEE Software, pp. 59-65 (May 1990).

7) 下村隆夫: Program Slicing技術とテスト，デバッグ，保守への応用，情報処理学会誌，Vol. 33, No. 9, pp. 1078-1086 (1992).

8) Podgurski, A. and Clarke, L. A.: A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance, IEEE Transactions on Software Engineering, Vol. 16, No. 9, pp. 965-979 (September 1990).

9) 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).

10) Weiser, M. and Lyle, J.: Experiments on Slicing-Based Debugging Aids, Empirical Studies of Programmers, Ablex Publishing Corporation, pp. 187-197 (1986).

11) Hausler, P. A, Pleszkoch, M. G., Linger, R. C. and Hevner, A. R.: Using Function Abstraction to Understand Program Behavior," IEEE Software, pp. 55-63 (January 1990).

12) Ott, L. M. and Thuss, J. J.: The Relationship between Slices and Module Cohesion, 11th ICSE, pp. 198-204 (May 1989).

13) Weiser, M.: Programmers Use Slices When Debugging, CACM, Vol. 25, No. 7, pp. 446-452 (July 1982).

14) Weiser, M.: Program Slicing, IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 352-357 (July 1984).

15) Leung, H. K. N. and Reghbati, H. K.: Comments on Program Slicing, IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, pp. 1370-1371 (Dec. 1987).

16) Bergeretti, J. F. and Carre, B. A.: Information-Flow and Data-Flow Analysis of while-

Programs, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp. 37-61 (January 1985).

17) 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 (January 1990).

18) Jiang, J., Zhou, X. and Robson, D. J.: Program Slicing For C - The Problems In Implementation, Proceedings of Conference on Software Maintenance-1991, pp. 182-190 (Oct. 1991).

19) Lyle, J. R. and Binkley, D.: Program Slicing in the Presence of Pointers, SERF'93, pp. 255-259 (Nov. 1993).

20) Venkatesh, G. A.: The Semantic Approach to Program Slicing, SIGPLAN NOTICES, Vol. 26, No. 6, pp. 107-119 (June, 1991).

21) エイホ，ウルマン著，土居訳：コンパイラ，培風館 (1986).

22) Icuma, N. T., Cheng, J. and Ushijima, K.: Decomposition Slicing Based on Program Dependence Graph, 情報処理学会第45回（平成4年後期）全国大会, 5T-3 (Oct. 1992).

「II 部　分散処理システム障害解析支援方式の研究」

1) Ehud. Y. Shapiro: Algorithmic program debugging, The MIT Press (1982).

2) H. Takahashi, E. Shibayama: PRESET - A Debugging Environment for Prolog, Logic Programming Conference, Tokyo, pp. 90-99 (1985).

3) N. Shahmehri, M. Kamkar, and P. Fritzson: Semi-automatic Bug Localization in Software Maintenance, Proceedings of Conference on Software Maintenance, pp. 30-36 (Nov. 1990).

4) P. Fritzson, T. Gyimothy, M. Kamkar, and N. Shahmehri: Generalized Algorithmic Debugging and Testing, ACM SIGPLAN Notices, Vol. 26, No. 6, pp. 317-326 (June 1991).

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

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

7) Bogdan Korel and Janusz Laski: 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，情報処理学会誌，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，情報処理学会論文誌，Vol. 33, No. 4, pp. 501-511 (1992).

10) Mark Weiser: Programmers Use Slices When Debugging, CACM, Vol. 25, No. 7, pp.

446-452 (1982).

11) Mark Weiser: Program Slicing, IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 352-357 (1984).

12) Shan Horwitz, Thomas Reps, and David Binkley: Interprocedural Slicing Using Dependence Graphs, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 26-60 (1990).

13) Mark Weiser and Jim Lyle: Experiments on Slicing-Based Debugging Aids, Empirical Studies of Programmers, Ablex Publishing Corporation, pp. 187-197 (1986).

14) J. R. Lyle and M. Weiser: Automatic Program Bug Location by Program Slicing, The Second International Conference on Computers and Applications, pp. 877-883 (June 1987).

15) B. Korel and J. Laski: Dynamic Program Slicing, Information Processing Letters, Vol. 29, No. 10, pp. 155-163 (Oct. 1988).

16) B. Korel and J. Laski: Dynamic Slicing of Computer Programs, J. Systems Software, Vol. 13, pp. 187-195 (1990).

17) Hiralal Agrawal and Joseph R. Horgan: Dynamic program Slicing, ACM SIGPLAN Notices, Vol. 25, No. 6, pp. 246-256 (1990).