ソフトウェア工学　131−7
（２００１．６．１）

少ない回数で行える分割検証支援ツールの開発

高橋 光弘，太田 剛，酒井 三四郎

静岡大学

プログラマは，多くの時間をデバッグ作業に費やす．一般的なデバッガは様々な補助を提供するが，複雑で難しい作業はプログラマが行わなければならない．プログラマは，プログラムを実行させた結果，どの命令が出力された変数の値に影響を与えているか，制御を決定した変数の値が何であるかを，見つける必要がある．

デバッグ技法の１つに分割検証法がある．ただし，この技法には多くの問題がある．例えば，実行履歴が長いと分割回数が増えたり，分割位置によっては検証しにくい場合があるといったことである．そのため，プログラマにとっては煩わしい．

本研究では，この分割検証法を支援するデバッグツールを紹介する．このツールには，動的スライシング技術が用いられている．

Development of a debugging tool for improved verification by division method

Mitsuhiro Takahashi, Tsuyoshi Ohta, Sanshiro Sakai

Shizuoka University

Programmers spend considerable time to debug their codes. General debuggers provide some help but the tasks are still complex and difficult. Programmers must perform many tasks manually that the tools could perform automatically, such as finding which statements in the program affect the value of an output variable for a given test case, and what is the value of a given variable at the last reached control statement.

“Verification by division” is one of a debugging method. But this method has many problems. For example, the longer log of execution we have, the more number of times of division we need, and it isn’t always easy to verify correctness of execution at any given execution point. So it is troublesome for programmers to debug their codes with this method.

In this paper, we present a debugging tool to support “verification by division” method, based on dynamic slicing technique.

1 はじめに

ソフトウェア開発において，プログラマは，まず目的の機能を達成するためのアルゴリズムを考え，それをソースコード上に命令として記述する．そして，記述したプログラムが，自分の意図したとおりに動くかどうかを確認する．プログラムを実行した結果，思いどおりの動作をしなかったり，期待した出力が得られなかった場合，プログラマはその原因，つまりバグの原因を突き止める必要がある．

バグの原因を突き止めるために，プログラマはデバッガなどを用いてデバッグ作業を行う．デバッガには，プログラムのある実行時点の変数情報などをチェックするブレークポイント機能や，１命令ずつ実行してプログラムの実行状態を確認するステップ（トレース）実行機能などがあり，プログラマはデバッガのそのような機能を用いてデバッグ作業を行う．

プログラマは変数に関する依存関係をたどり，それぞれの命令が正しい記述と値で実行されているかどう

かを調べることによって，バグの原因を突き止めている．それを図１を用いて説明する．プログラマが，命令 $sum = c + d$の変数 sumの値が間違っていることに気付いたとする．プログラマはまず，変数 sum の値を決定している命令の記述が正しいかどうか確認する．間違っていれば命令を書き直し，間違っていなければ，その命令で使われている変数の値を確認する．変数 c の値が間違っていて変数 dの値が正しいと分かった場合，変数 cの値を決定した命令を探し出す．そして，同様の作業を探し出した命令に対して行う．このような作業を繰り返すことによって，バグの原因を突き止める．

こういった，命令間の依存関係を計算する技法として，プログラムスライシングがある[1]．プログラムスライシングとは，ある命令に使用されているある変数を指定した場合，その変数の値に影響を与える命令をソースコードから抽出してくれるというものである．

一般的なデバッガには，このような，命令間の依存関係をたどるための機能は無いが，そのような機能を持つデバッガとして，SPYDER がある[2]．SPYDER を用いたデバッグ手順は，まず，ある変数に関するプログラムスライシングを計算し，その中に含まれる命令の中で怪しいと思われる命令の実行状態を調べる．ただし，依存関係に含まれている命令が正しいかどうかの判断は，プログラマが行う．また，ブレークポイントやトレースポイントで得られる情報（変数情報など）の何について調べるかもプログラマが考え，確認作業を行っている．

まとめると，デバッグ時にポイントとなる作業は 2 つある．

- どの命令の実行状態を調べるかを決めること．
- 選んだ命令のどの箇所を調べるかを決めること．

これらは，プログラマの経験的な部分に頼るところが多く，作業効率を大きく左右する．経験豊かな人はこの決定が適確であり，そのためデバッグ作業が早く，逆に経験が乏しい人はこの決定を適確に行えないためにデバッグ作業に手間がかかることになる．

この２つの点を克服するデバッグ技法として，分割検証法がある[3]．この技法は，まず，プログラムの実行履歴から命令間の依存関係を計算する．次に，その依存関係を用いて，バグの原因が含まれるフローグラフを作成する．そして，それを２分探索するようにして分割し，その分割点を横切るデータフローや制御フローが正しいか正しくないかを，プログラマが判断する．この作業を繰り返すことにより，最終的にバグの原因を突き止める技法である．分割点はシステムにより自動的に決められ，分割点で確認するフロー箇所もシステムが提供するため，プログラマはシステムが提供した実行状態の正誤を判断していくだけでよい．

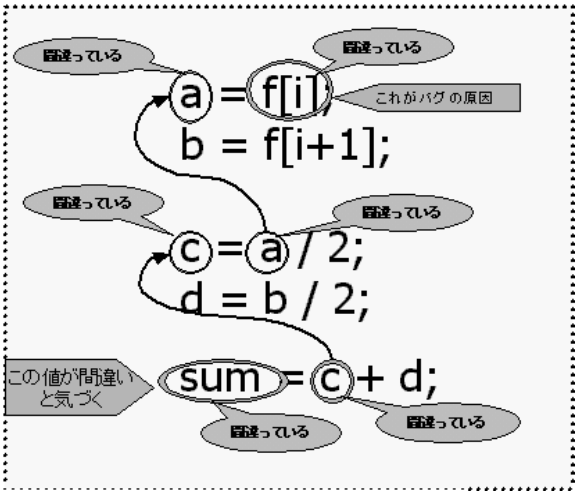


図1:バグの原因をたどる手順

しかし，分割検証法には２つの大きな問題点がある．

- フローグラフに含まれる実行時点の数が多いと，分割回数が増える可能性がある．
- 分割点によっては，調べるフローの数が多かったり，正誤を判断しにくい場合がある．

このような理由が原因で，分割検証を行う時のプログラマへの負担が大きくなってしまう．

そこで本研究では，分割検証法の欠点を克服するための手法を２つ提案する．これらの手法を導入することにより，分割検証法を用いるプログラマへの負担を軽減させることが期待できる．

(1) 動的スライスによる制御命令のチェック

分割検証を行う前に動的スライスを確認する．もし間違った制御が行われている部分が見つかれば，その箇所について新たにフローグラフを作成する．また，制御が正しく実行されている制御命令をチェックし，フローグラフを作成する時にはその命令に関する依存関係をたどらないようにする．

これらの作業の結果，分割検証対象のフローグラフのノード数を減らすことができる．

(2) 分割点の位置の工夫

分割点を，if や while や for などの制御ブロックの外に置くようにする．その結果，分割点で確認するフローの制御に関する部分を減らすことができる．

この後，まず第2章では動的プログラムスライシングを紹介し，第3章では分割検証法を紹介する．次に第4章では，提案した2つの技法と本ツールの紹介する．そして第5章では，ツールの評価と考察を述べ，最後に第6章では，本研究のまとめを述べる．

2 動的プログラムスライシング技術

プログラムスライシングとは，プログラム内の命令間の依存関係を明らかにする技術である．プログラムスライシングには，静的スライシング[1]と動的スライシング[4]がある．静的スライシングとは，ソースコードに書かれている命令間の依存関係を解析することにより，ある変数に影響を与える可能性のある命令を抽出することをいう．それに対し動的スライシングとは，まず，一度プログラムを実行させて，どのような命令がどういった値で実行されていったかという実行履歴を記録する．そして，その実行履歴に書かれている命令間の依存関係を解析することによって，ある変数に実際に影響を与えた命令を抽出することをいう．また，静的スライシングによって抽出された命令のことを静的スライス，動的スライシングによって抽出された命令のことを動的スライスと呼ぶ．

動的スライスを定義するためには，まず，どの実行点のどの変数に関して動的スライスを求めるかを決める必要がある．その動的スライス計算を開始するための基準となるものを，スライシング基準と呼ぶ．

プログラムPのスライシング基準$C=(x, r, V)$とは，次の3つの組をいう．

(1) xはプログラムPに与えた入力．

(2) rはプログラムPに入力xを与えたときの実行系列における実行点．

(3) VはプログラムP内の変数の部分集合．

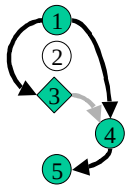
プログラムPのスライシング基準$C=(x, r, V)$に関する動的スライスとは，以下の条件を満たす，任意の実行可能なプログラムP'である．ただし，プログラムPに入力xを与えたときの実行系列をEX，プログラムP'に入力xを与えたときの実行系列をEX'とする．

(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である．

(2) 実行系列EX'において，ある実行時点r'が存在し，実行系列EX'の実行時点r'までの部分列が，実行系列EXの実行時点rまでの部分列から，P'を得るためにPから削除された命令を除外したものに等しい．

```
1   scanf(“%d %d“,x,y);
2   max = x;
3   if ( x < y ) {
4       max = y;
    }
5   printf(“%d”,max);
```

サンプルプログラム




```
1   scanf(“%d %d”,x,y);     { x = 2, y = 3 }
2   max = x;                { max = 2 }
3   if ( x < y ) {          { if ( 2 < 3 ) }
4       max = y;            { max = 3 }
5   printf(“%d”,max);       { printf(3); }
```

図2：実行時点5の変数*max*に関する動的スライス

(3) 任意の変数$v \in V$に対して，実行系列EX'において実行時点r'の直前における変数vの値が，実行系列EXにおいて実行時点rの直前における変数vの値に等しい．

また，図2の左に示すような，実行点をノードとし，依存関係をアークとして表したものをフローグラフという．

なお，動的依存関係やフローグラフの計算方法は，本稿では省略する．

3 分割検証法

実行時点r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，まず，実行時点rにおける変数vに関するフローグラフを作成する．次に，そのフローグラフの中に適当な実行時点を決め，その実行時点を横切るデータ依存や制御依存などの命令間の依存関係が正しいかどうかを判断する．もし，間違ったフローが見つかれば，その実行時点より前にバグの原因があることになり，間違ったフローがなければ，その実行時点より後にバグの原因があることになる．このような，フローグラフの分割を繰り返すことによりバグを検出するアルゴリズムを，分割検証法[3]という．

しかし，分割対象となるフローグラフにバグの原因となるものが含まれていなければ，分割検証を行ってもバグの原因を特定することはできない．動的スライスを求める場合には，定義依存関係と制御依存関係を用いてフローグラフを作成するが，これだけではバグの原因を含まない場合がある．そのため，分割検証法にはクリティカルフローグラフという，特殊なフローグラフを用いる．動的スライスを求める場合には，定義依存，制御依存の2つの依存関係を用いてフローグラフを作成した．一方，クリティカルフローグラフでは，定義依存関係，制御依存関係，制御定義依存関係，

分岐設定漏れ依存関係，配列設定漏れ依存関係，の５つの依存関係を用いてフローグラフを作成する[3].その作成の仕方は，本稿では省略する.

4 分割検証支援ツール

分割検証法の欠点には，以下の２つがあった.

- フローグラフに含まれる実行時点の数が多いと，分割回数が増える可能性がある.
- 分割点によっては，調べるフローの数が多かったり，正誤を判断しにくい場合がある.

この章では，分割検証法の欠点を克服するために提案した２つの手法について述べる.

4.1 動的スライスによる制御命令のチェック

この作業の目的は，分割検証を行う前に，できるだけ検証対象となる実行点の数を減らすことにある．そのために，プログラマは分割検証を行う前に動的スライスを確認する.

```
Node getDependency( これから依存関係をたどる実行時点，exeNum )
{
   // まず，exeNumへの４つの依存関係をたどり，実行時点を得る.
   DefDef = getDef(exeNum);          // exeNumへの定義依存，DefNumを得る.
   CtlDef = getCtlDef(exeNum);       // exeNumへの制御定義依存，CtlDefを得る.
   OmsCond = getOmsCond(exeNum);     // exeNumへの分岐設定漏れ依存，OmsCondを得る.
   OmsArray = getOmsArray(exeNum); //exeNumへの配列設定漏れ依存関係，OmsArrayを得る.

   // 以上より，exeNumへ依存関係がある実行時点Dependは，
   Depend = { 実行時点x | x ∈ DefNum∪CtlDef∪OmsCond∪OmsArray};

   // ここで，正しいと判断された制御命令の実行時点TrueNumを，Dependから取り除く.
   NewDepend = { 実行時点x | x ∈ Depend - Depend∩TrueNum };

   // exeNumに依存関係が発生している実行時点の集合，NewDependを返す.
   return NewDepend;
}
```

図4.1:正しい制御命令の除去アルゴリズム

```
int get_DivPoint( int size ){
    // まず，検証対象の実行点の真中を基準とする.
    int span 確認範囲の値

    // まず、確認箇所数の調べる範囲を設定する。
    int centerIndex = 検証対象の実行点の真中の位置;
    int fromIndex = centerIndex - span;  // この位置から、
    int toIndex = centerIndex + span;    // この位置まで検索する。

    // つぎに，centerIndexの位置の分割点を確認箇所の最小個数minSizeとする
    int minSize = getCrossFlow( centerIndex );  // 指定した位置の確認箇所の個数を得る
    int minIndex = centerIndex;

    // fromIndexからcenterIndexまで探す。
    int size;
    for( int i = fromIndex ; i < toIndex ; i++ ){
        size = getCrossFlow( i );
        if( size < minSize ){
            minSize = size;
            minIndex = i;
        }
    }

    // そして，確認箇所数が一番小さい分割点の位置を返す
    return minIndex;
}
```

図4.2:分割点の工夫アルゴリズム

もし動的スライスの中に間違った制御が行われている命令が見つかれば，その命令について新たにフローグラフを作成する．つまり，現在のスライシング基準より前の位置に，スライシング基準を設定する．スライシング基準を実行時点の早い段階に設定すればするほど，クリティカルフローグラフに含まれる実行時点の数を減らすことができる.

また，制御が正しく実行されている制御命令をチェックし，フローグラフを作成する時にはその命令に関する依存関係をたどらないようにする．ここで，制御が正しく実行されている制御命令とは，制御命令が正しく記述してあり，かつ，そこに現れるすべての変数が正しい値で実行されるとプログラマが判断した制御命令のことをいう．分割検証を行う前に，制御が正しく実行されている命令を検証対象から取り除く．これは，dicing[5]の考え方に基づいている．この作業によって，クリティカルフローグラフに含まれる実行時点の数を減らすことができる．実行時点の数が減れば分割回数が減り，また，分割点を横切るデータフローや制御フローの数が減る可能性が高い.

実行時点の依存関係をたどるときに，正しいと判断した制御命令に含まれる実行時点の除去アルゴリズムを図 4.1 に示す.

4.2 分割点の位置の工夫

この作業の目的は，分割検証作業において，個々の分割点で確認すべきフローの数を減らすことにある．そのために,分割点を if や while や for などの制御ブロックの外に置くようにする．その結果，制御に関する確認箇所を減らすことができる.

普通，for や while などの繰り返し文の内で分割する場合，分割点では制御依存関係が重複して発生しているために検証すべき制御命令が多くなっている．くわえて，ループの制御のみに関する変数などもあるため，検証作業が大変である.

ある分割点を基準としてその周辺を探した時，確認すべきフローの数が局小となる分割点がある．この分割点は，そこが制御に関するフローが少なくなってい

る分割点である傾向が強い．つまり，そのような分割点が制御の区切れである．本ツールでは，この場所を新たな分割点とする．このアルゴリズムを図 4.2 に示す．

4.3 ツールの概要

ツールの概要を図 4.3 に示す．まず，デバッグ対象プログラムを実行して実行履歴ファイルを得る[1]．次に，ソースファイルとその実行履歴を，ツールに入力として与える．ツールは実行履歴を用いて命令間の依存関係の計算，動的スライシング，クリティカルスライシング，また，分割検証法が実行できる．プログラマはこのツールを操作してデバッグ作業を行う．

このツールには図 4.4 に示すように，4 つの内部ウィンドウがある．

- Source Information（図 3.4 の左上）
- Variable Information（図 3.4 の左下）
- Record Information（図 3.4 の右上）
- Division Information（図 3.4 の右下）

それぞれのウィンドウの役割を以下に述べる．

● Source Information ウィンドウ

プログラマは，このウィンドウに表示されるソースコードを見てスライス基準を決めたり，正しい制御命令のチェックを行う．このウィンドウの内部は表になっており，1 番目の列“行”はソースコードの行番号，2 番目の列“正誤”は制御命令の正誤，3 番目の列“S”は動的スライス，4 番目の列“命令”はソースコードの命令が入る．

動的スライスを計算したい命令を選んで右クリックするとその命令のスライス基準のポップアップが現れ，動的スライスを求めることができる．そして，スライスを計算した結果，列“S”に記号“⇒”が現れた命令は，その命令がスライスに含まれることを意味する．

また，スライスに含まれる制御命令の中で，正しいと思われるものと同じ行の列“正誤”をクリックすると，記号“○”現れる．そして，もう一度スライスを計算し直すと，チェックした制御命令への依存関係を省いたスライスが計算される．

[1]現在は，実行履歴をファイルへ出力するための命令を，手動でデバッグ対象プログラムにあらかじめ埋め込んでおくことによって行う．

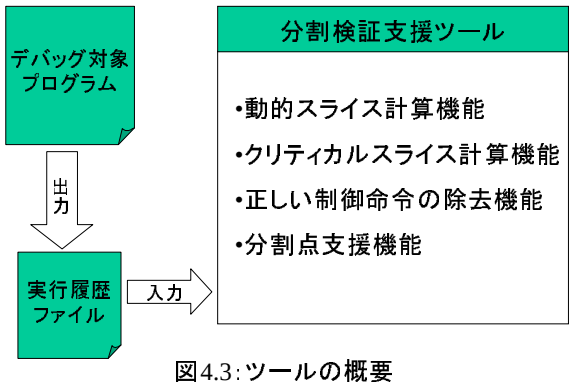


図4.3:ツールの概要

● Record Information ウィンドウ

このウィンドウには，クリティカルスライスに含まれる実行時点の命令が表示される．このウィンドウの内部は表になっており，1 番目の列“行”はソースコードの行番号，2 番目の列“正誤”は命令の正誤，3 番目の列“P”は分割点，4 番目の列“実”は実行履歴の実行番号（その命令が，何番目に実行されたかというもの），4 番目の列“命令”にはソースコードの命令が入る．

Source Information ウィンドウでスライス基準を選択すると，同時にクリティカルスライスも計算され，そのスライスに含まれる命令の一覧がこのウィンドウ内に表示される．これらの命令を対象に，分割検証が行われる．

分割検証を行う過程で，正しいと判断されたり，検証を行う必要が無くなった実行時点の命令は，列“正誤”のセルに記号“○”が現れる．そのため，分割検証がどの程度進んでいるかを見ることができる．

● Variable Information ウィンドウ

このウィンドウでは，他のウィンドウで指定された命令の変数情報を確認することができる．for 文や while 文など，その命令が複数回実行された場合でも，ウィンドウ下の ComboBox で何回目かを選択することができる．

● Division Information ウィンドウ

このウィンドウでは分割検証を行う．“次の分割点へ”というボタンをクリックすると，4.2 節に示した方法によって自動的に分割点が決定され，その分割点を横切るデータフローと制御フローに含まれる命令が抽出され，表示される．プログラマは，検証項目を選択し，右クリックしてポップアップを表示させ，正しければ“true”を選択，間違っていれば“false”を選

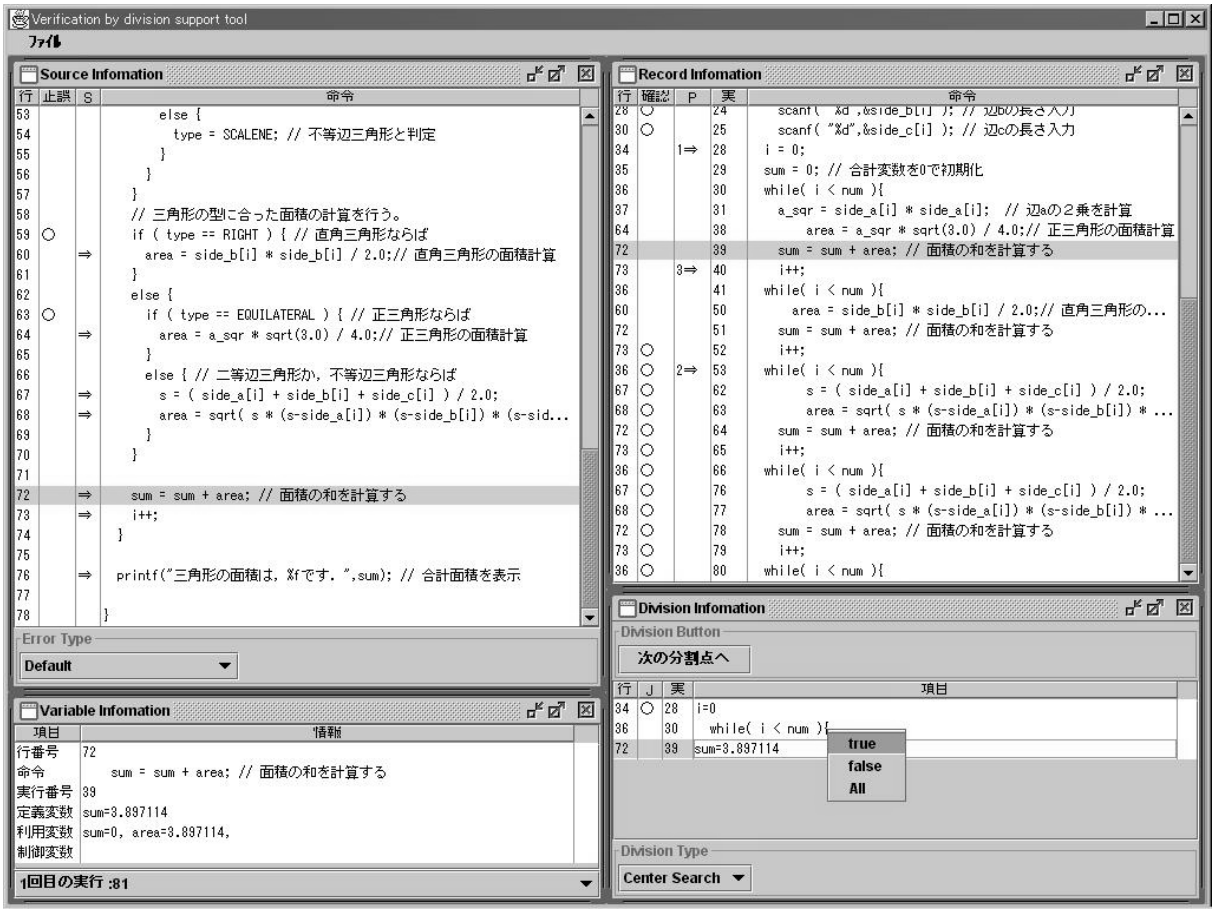


図4.4:ツールの画面

択する．もし間違っていれば，Record Information ウィンドウの分割点より下の検証項目の列“確認”のセルに確認済みの記号“○”が現れ，もし，すべての項目が正しいかった場合は，分割点より上の検証項目の列“確認”のセルに記号“○”が現れる．そして，プログラマは“次の分割点へ”というボタンをクリックし，新しい分割点を決定して，最終的にバグを見つける．

5 評価と考察

本研究で実装したツールの評価実験を行い，有用性を確かめる．

5.1 実験方法

実験計画法[6]に基づいて 3 つの因子を用意した．

- デバッグするプログラム（因子 A）
  A1: プログラム a（45 行程度．履歴命令は 50 個）
  A2: プログラム b（60 行程度．履歴命令は 89 個）
  A3: プログラム c（75 行程度．履歴命令は 81 個）
- デバッグ方法（因子 B）
  B1: 分割検証
  B2: 分割検証＋制御命令のチェック
  B3: 分割検証＋分割点の工夫
- 被験者の種類（因子 C）
  C1: デバッグが不得意な人
  C2: デバッグが得意でも不得意でもない人
  C3: デバッグが得意な人

まず，アンケートにより因子 C の種類毎に被験者を 3 人集める．α さんはプログラム A1 をデバッグ方法 B1 でデバッグし，β さんはプログラム A1 をデバッグ方法 B2 でデバッグし，γ さんはプログラム A1 をデバッグ方法 B3 でデバッグする．同じようにして，プログラム A2，A3 に関してもデバッグを行う．ただし，同じ人が同じデバッグ方法でデバッグは行わないよう組み合わせる（例を表 5.1 に示す）．そして，ツールを用いてデバッグを行い，バグが見つかるまでに分割点で確認したフローの数（以下，これを確認箇所数と呼ぶ）を記録する．

5.2 実験結果

表 5.2 は実験結果を示したものであり，セル内の数字はバグが見つかるまでの確認箇所数である．そして，これを分散分析した結果を表 5.3 に示す．表 5.3 の列 F の値が，4.46 を超えると危険率 5%で有意，8.65 を超えると危険率 1%で有意であると言える．

表 5.3 から分かるように，プログラムの種類とデバッグ方法の因子が，確認箇所数に大きく影響を与えると判定されている．

プログラムの種類が，確認箇所数に大きく影響を与えることは当然である．なぜなら，プログラムが異なれば実行履歴も異なるために，バグの発見手順が大きく異なってくるからである．

デバッグ方法が，危険率 1%で有意な因子であると判定されたことは，制御命令のチェックや分割点の工夫が確認箇所数に大きく貢献していることの証明であり，本研究の目的を達成している．この詳細を図 5.1 に示す．縦軸の確認個所数の和というのは，今回の実験で得られた，それぞれのデバッグ方法でデバッグした場合の確認箇所数の合計である．制御命令のチェックや分割点の工夫を行った場合の方が，確認箇所数が普通に分割検証する場合の 4 分の 3 程度で済んでいる．

実験前の段階では，正しい制御命令を省くことができれば，デバッグ方法 B1 よりも B2 の方が確認箇所数は減少すると予想していた．しかし，表 5.2 のプログラム A2 に関する実験結果を見て分かるように，デバッグ方法 B2 の時の確認箇所数が，B1 の時よりも増えている．これは，分割検証対象の実行点の数が少なくても，分割点の場所が悪かったためだと考えられれる．ただし，今回の実験で用いたデバッグ用のプログラムが小規模なものであるために，このような結果になったとも考えられる．

また，デバッグ方法と被験者の種類に着目してみると，デバッグ方法が B1 や B3 の時の検証箇所数の値は，被験者の種類は違っても同じ値になっている（表 5.2 を参照）．これは，分割検証対象の実行点の数が同じであれば，バグが見つかるまでの手順は誰が行っても同じになることを示している．

6 おわりに

評価実験から，分割検証法を用いたデバッグにおいて，提案した 2 つの技法によって確認箇所数が減少することが分かり，プログラマへの負担が軽減されることが確認できた．

表5.1:組み合わせの例

	B1	B2	B3
A1	α	β	γ
A2	γ	α	β
A3	β	γ	α

表5.2:実験結果

プログラムA1

	B1	B2	B3
C1	20	16	12
C2	20	10	12
C3	20	12	12

プログラムA2

	B1	B2	B3
C1	18	21	17
C2	18	21	17
C3	18	11	17

プログラムA3

	B1	B2	B3
C1	30	24	22
C2	30	24	22
C3	30	14	22

表5.3:$L_{27}(3^{13})$の実験に対する分散分析表

要因	SS	f	V	F	判定
M	9633.3333	1	9633.3333	—	
A	416.0000	2	208.0000	21.3445	**
B	192.6667	2	96.3333	9.8855	**
C	34.6667	2	17.3333	1.7787	
$A\times B$	34.0000	4	8.5000	0.8723	
$A\times C$	16.0006	4	4.0002	0.4105	
$B\times C$	69.3334	4	17.3334	1.7787	
e	77.9993	8	9.7449	—	
計	10474.0000	27	—	—	

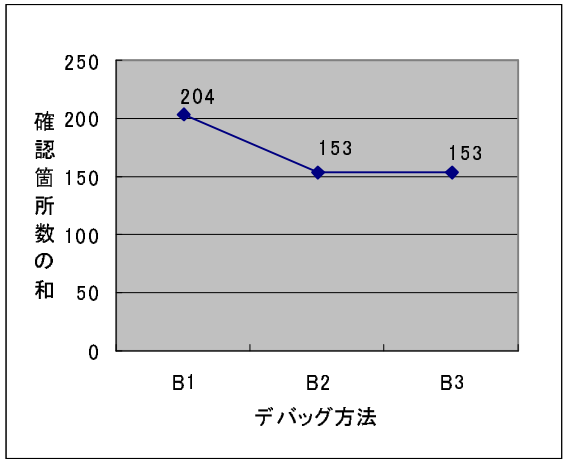


図 5.1:因子 C に関するグラフ

しかし，今回の評価実験でさらに検討すべきことがいくつか見つかった．制御命令をチェックしてから分割検証を行う場合とそのまま分割検証をした場合とでは，確認箇所数には違いが出るが，それぞれの作業を総合してみた場合，どちらの方の効率がよかったかは判断しにくい．また，分割検証対象に含まれる実行点の種類や数が変われば，検証作業における確認箇所の数や種類も変わる．そのため，分割検証対象の実行点

の数が少し減ったからといって，確認箇所数が減るということは必ずしも言えないことが分かった．

ところで，分割検証法によるデバッグにおいてまだ解決すべき課題がある．それは，分割点の確認箇所の中で，正しいか正しくないか判断できないものがあった場合，どう対処するかという問題である．今回の評価実験でデバッグに使用されたプログラムは，小さかったために全部の変数を把握することが容易であったが，大きなプログラムをデバッグする場合，プログラマが正誤を判断しにくいものが出てくることは当然である．そのため，正誤が判断できないものがあっても，バグの原因を特定することができるような工夫が必要である．

また，本ツールは分割点をシステムが自動的に決めるだが，その分割点が，プログラマにとって必ずしも検証しやすい場所だとは限らない．プログラマにとって検証しやすい場所は，プログラマ自身が知っているはずなのだから，判断しやすい位置で分割検証するような工夫が必要である．例えば，プログラマが，あらかじめソースコードに分割してほしい位置の候補となるような箇所に何らかの印をつけておき，デバッグ時にこのツールがその部分を探して，分割するようなことを行えば良い．

このような課題を解決することができれば，分割検証法を用いたデバッグ技法は，さらに有効なデバッグ技法となることが期待できる．

7 参考文献

[1] M.Weiser,“Program slicing,” IEEE Transactions on Software Engineering, Vol.SE-10, no.4, pp.352- 357, 1984.

[2] H. Agrawal and R. A. Demillo and Eugene-H. Spafford, “Debuging with Dynamic Slicing and Backtracking,” Software—Practice and experience, VOL. 23(6), 589-616, June 1993.

[3] 下村 隆夫, “プログラムスライシング技術と応用,” 共立出版, 1995.

[4] B. Korel and J. Laski, “Dynamic Program Slicing,” Information Processing Letters, Vol.29, No.10, pp.155-163, Oct. 1988

[5] M. Weiser and J. Lyle, “Experiments on Slicing-Based Debugging Aids,” Empirical Studies of Programmers, Ablex Publishing Corporation, p187-197, 1986.

[6] 中村 義作, “よくわかる実験計画法,” 近代科学社, 1997.