スライシングを用いたデバッグ支援ツールに関する一考察

工藤　英男・中井　伸郎*・内田　眞司**

Program Debug Supporting Tool by Program Slicing

Hideo KUDO, Nobuo NAKAI* and Shinji UCHIDA**

ソフトウェア開発において，最も時間がかかる作業がデバッグであり，全工程の５割程度を占めると言われている．しかし，現在でも，デバッグを効率的に行う完全な方法論は提案されておらず，実務上ではプログラマの技量に頼っているのが実状である．そこで，本研究では，デバッグにかかる時間を短縮し，短期間でのソフトウェア開発を可能とするため，スライシング技術を利用したデバッグ支援ツールの実現と有用性について考察する．

１．はじめに

ソフトウェアのライフサイクルにおいて，保守工程を除いた設計・開発工程の中でデバッグにかかる費用は，全体の開発コストの約40～50%といわれている．そして，デバッグに関する様々な方法論が提案されているが，これらの方法論は完全とはいえない．

発生したバグを効率よく除去する方法として，1982年にMark Weiserによって提案されたスライシング技術[1]がある．スライシングはプログラムの命令間の依存関係を明らかにする技術である．依存関係を求めデバッグ時に読みとるコードの量を少なくすることで，プログラマのプログラム理解を助けデバッグの時間を短縮できる[2]．

スライシングにおいては，依存関係を効率よく求める方法[3,4]，初心者プログラマにスライシングを用いてバグプログラムをデバッグさせ，バグの発生パターンを学ばせる方法[5]や，複数のスライスの差分を取り，読みとるコードの量を減らすデバッグ法の研究[6]がされている．

筆者らは，Pascalで表記したプログラムについてのスライシングの検討[7]やプログラマはどのようにバグの原因を特定しているか，デバッグ中にどのようなことを考えているかをモデル化する研究[8,9,10]を進めてきた．

本稿は，スライシングを用いたデバッグ手法の研究[11]を再構成したものである．以下，２章ではスライシングについて説明し，３章ではスライシングを利用したデバッグ支援ツールを提案する．４章ではデバッグ支援ツールの仕様について述べ，５章では実現した支援ツールの評価実験と結果及び考察を述べる．

＊広島大学工学部
＊＊近畿大学工業高等専門学校

２．スライシング

この章では，スライシングの概要とスライシングの例について述べる．

2.1　スライシングの概要

スライシングとは，解析対象となるプログラムPからスライスを求める操作のことである．プログラムPのスライスとは，以下の条件を満たす実行可能なプログラムP'のことを表す．

P'はPから０個以上の命令を削除して得られるプログラムである．入力ｘに対してPが停止するなら，P'も入力ｘにより停止する．

スライスを求めるためには，プログラムP内のどの命令のどの変数に対して，スライシングを行うかを入力する必要がある．これをスライシング基準といい，Ｃ＝(u,V)と表す．ｕはプログラムP内のある命令を，Ｖはプログラム P内の変数の部分集合を意味する．その入力されたスライシング基準を基に，ｕ中のＶに対するデータ依存関係と制御依存関係を求めることでスライスが求められる．

なお，スライシングには動的スライシングと静的スラ

イシングの２種類が存在するが，本稿では静的スライシングを採用し，以下，単にスライシングと呼ぶ．

2.2　スライシングの例

具体例として，Ｃ言語で表記した解析対象のプログラムを図１に示す．

```
        void main(void) {
          File  *fp;
          int IsWorld, lines, words, characters;

 1:     IsWord=0;
 2:     lines=0;
 3:     words=0;
 4:     characters=0;
 5:     if ((fp=fopen(" input.txt" ," r" ))==NULL)
 6:        return(-1);
 7:        while((c=fgetc(fp))!=EOF) {
 8:           charcters++;
 9:           if (c='\n')
10:              lines++;
11:           if( c==' ' I c='\t' I c==" \n)
12:                 IsWord=0;
13:           else if (IsWord==0) {
14:                 IsWord=1;
15:                words++;
                   }
          }
16:       printf(" %dline,%d,word,%d,character" ,
                   lines,words,characters);"
       }
```

図１　解析対象のプログラムＰ

図１において，１行につき１つの命令を記述しているが，関数と制御文の処理範囲を表す’{’と’}’は，命令とみなさない．また，行の左にある’：’以前の文字は命令の番号を表しているが，関数の宣言，初期化処理をしていない変数の宣言，コメントには番号をふらない．

そこで，命令16に存在する変数charactersに対してスライシングを行うと，図２に求まったスライスを示す

つまり，スライシング基準はC=(16,characters)となり，スライスP’={4,5,7,8,16}が求まる．結果として，もともと16行あったプログラムが，charactesに対しては図２のように5行へと短縮される．

```
        void main(void) {
 4:     characters=0;
 5:     if ((fp=fopen(" input.txt" ," r" ))==NULL)
 7:        while((c=fgetc(fp))!=EOF) {
 8:           charcters++;
          }
16:       printf(" %dline,%d,word,%d,character" ,
                   lines,words,characters);"
       }
```

図２　求まったスライスP’

命令16の実行に影響を与える可能性のある命令，すなわちcharactersの計算に関連する部分だけが抽出されるため，プログラムの理解が容易となる．

また，スライスとして求まった命令の集合は，実行可能な部分プログラムになるので，変数charactersに関しては同じ入力ならば，もとのプログラムと同じ値を出力することが可能である．

3．スライシングを利用したデバッグ法

この章では，スライシングを利用したデバッグ支援ツールで用いる解析開始位置，機能の定義，機能の分割，重み付けについて述べる．

3.1　解析開始位置

スライシングでは，解析するプログラムに対してスライシング基準を与える必要があった．提案するデバッグ法も，どの命令から解析するかという情報を与える必要がある．これを解析開始位置といい，A＝(u)と表す．uはスライシング基準と同じで，解析するプログラムＰに存在する命令である．

3.2　機能の定義

与えられた解析開始位置に存在する変数の集合をＶ，存在する変数の数をｎ個とした時，Ｖ中の変数の一つをv_i(ｉ＝1, 2 ...n)と表す．ここで，スライシング基準をC_i =(u, v_i)として，ｉを１からｎまで変化させてスライシングする．求まったｎ個のスライスを機能f_iと呼ぶ．

図１のプログラムを例にする．解析開始位置が16行目だとすると，変数の集合Ｖは，

Ｖ＝{lines, words, characters}

となり，その個数は３である．この３つの変数に対してスライシングすると，

C_1(16,lines)：P_1’={2,5,7,9,10,16}＝f_1

C_2(16,words)：P_2’={1,3,5,7,11,12,13,15,16}＝f_2

C_3(16,characters)：P_3’={4,5,7,8,16}＝f_3

と，３つの機能が求まる．

3.3　機能の分割

前項の機能を分割し上下関係を持たせ，’位’と’子’について定義する．’位’は機能間の上下関係を表す数値で，数字が小さいほどその機能は上位である．

①前項で求まった３つの機能は最上位の機能であるとし，

level(f_1)=level(f_2)=level(f_3)=0　と表す．

②同じ’位’を持つ機能同士を比べて共通要素があるか調べる．もし存在するなら，その共通要素の集合を新

たに機能として定義する．この新たに定義された機能は，比較した２つの機能を親に持つ'子'といい，'子'の'位'は親よりも一つ下位の機能であると定義する．この操作を各機能間で共通要素がなくなるまで繰り返す．

③先程求まったf_1からf_3までそれぞれ比較すると，

$f_1 \cap f_2 = \{5,7,16\}$

$f_2 \cap f_3 = \{5,7,16\}$

$f_1 \cap f_3 = \{5,7,16\}$

$\therefore f_1 \cap f_2 \cap f_3 = \{5,7,16\} = f_4$

と新たな機能f_4が定義され，その'位'は，

$level(f_4) = level(f_1) + 1 = 1$　である．

④'子'を持った親は自分が持つ'子'の集合childに新たに定義した'子'を追加し，自分が持つ要素から'子'が持つ要素を削除する．

⑤機能が定義された段階では，その機能が持つchildは要素0(ϕ)である．

⑥先程求めた'子'の要素をf_1，f_2，f_3から削除し，childに'子'を追加する．

$f_1 = f_1 - f_4 = \{2,9,10\}$

$f_2 = f_2 - f_4 = \{1,3,11,12,13,15\}$

$f_3 = f_3 - f_4 = \{4,8\}$

$child(f_1) = child(f_2) = child(f_3) = \phi + f_4 = \{f4\}$

⑦これで機能間に共通要素がなくなったので，機能分割が完了したことになる．

3．4　重み付け

図３は，分割した機能を階層的に示したものである．

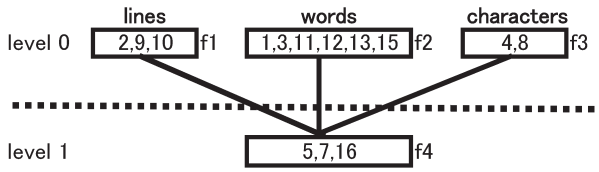

図３　機能の階層表示

四角形は機能を表し，その右側に機能の番号を，内側に機能が持つ要素を表す．図左のlevelという表示は，点線で区切られている行にある機能の'位'の高さ表す．機能と機能を結ぶ直線は，それらの機能間に親子関係がある事を表し，上側にある機能は親，下側にある機能は'子'である．そして，最上位(level 0)の機能を表す四角形の上にある変数名は，その機能が上に示す変数名のスライスを表す．

この図を例に重み付けを行うことにする．図１のプログラムで７行目の命令を間違って，

while((c=fgetc(fp))==EOF)

と記述したとする．この状態でプログラムを実行させると16行目の命令で，

0 line, 0 word, 0 character

と画面に出力される．

lines，words，charactersの全ての変数が期待した値を返さないので，解析開始位置をA＝(16)として，この３つの変数をバグの原因であると仮定を立てる．そして，それらの変数が指し示す機能f_1，f_2，f_3に重みを１ずつ割り振る．各機能の重みは，

$weight(f_1)=weight(f_2)=weight(f_3)=1$　と表す．

次に，重みを割り振られた機能は，自分が持つ重みを自分の全ての'子'に継承させる．この例ではf_1，f_2，f_3が持つ唯一の'子'であるf_4に重みを継承させることになる．'子'であるf_4の重みは，親から継承した重みの総和となり，

$weight(f_4) = weight(f_1) + weight(f_2) + weight(f_3)$

$= 1 + 1 + 1 = 3$　である．f_4は'子'を持っていないので，これで重み付けは完了である．

重み付けが完了したら，重みの高い機能が持つ要素からプログラムを参照していくと，バグを速く発見できる可能性がある．上の例では，重みが一番高いf_4から見ていくと，参照すべき命令は5,7,16である．結果的には，７行目の命令でwhile文の条件式が間違っていることに気付き，16行のプログラムをわずか３行見るだけで良くなる．

4．デバッグ支援ツール

本章では，スライシングと機能分割を行う支援ツールの仕様や各処理部の働きについて説明する．

4.1　支援ツールの概要と構成

デバッグ支援ツールの構成を図４に示す．図の四角形は処理部を表し，円柱はファイルを表す．処理部は，前処理部，解析部，機能分割部，視覚化部の４つの部分からなる．ファイルには入力となるＣ言語のソースプログラム，中間ファイル，解析結果ファイル，分割結果ファイルがある．ただし，ツールの解析対象を，「１つ以上のファイルから構成されていて，ANSI標準の文法に従って記述されたＣ言語で，コンパイルが正常に通るソー

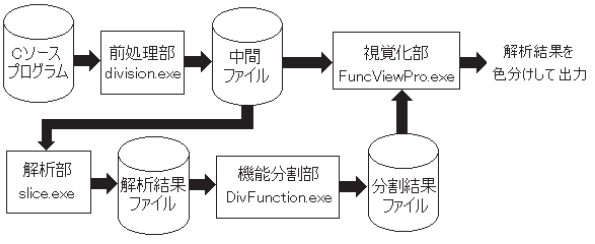

図４　支援ツールの構成

スプログラム」と定義する．

支援ツールは，ソースプログラムを読み込むと，前処理部で解析部が解析しやすい形式の中間ファイルを出力し，それを解析部が読み込みキーボードから入力された解析開始位置をもとに，解析開始位置に存在する全変数のスライスを求めて解析結果ファイルを出力する．次に解析結果ファイルを読み込んだ機能分割部が，各変数のスライスを基に機能分割を行い分割結果ファイルを出力する．最後に分割結果ファイルとソースプログラムを視覚化部が読み込み，疑わしい変数を入力することで，重み付けを行い，重みのあるコードを色付けて表示する．

4.2　各処理部の働き

以下に，４つの各処理部の働きについて述べる．

＜前処理部＞　解析部が解析しやすい形式の中間ファイルを生成するために，以下の３つの処理を行う．

・ プリプロセッサの処理を行う
・コメントを除去する
・プログラムを１命令ごとに分解する

プリプロセッサの処理では，複数ファイルで構成されるプログラムの場合，メインファイル以外のファイルを読み込み，条件コンパイル命令でコンパイルされない範囲を除去する処理を行う．プログラムを１命令ごとの分解は，人によって記述スタイルが違うので，スペース・タブ・改行文字を読み飛ばし，命令の部分だけを読み込んでテキストファイルに出力する処理である．

＜解析部＞　前処理部で生成された中間ファイルを読み込み，入力された解析開始位置から解析結果ファイルを生成するために，以下の３つの処理を行う．

・プラグラム中に存在する関数や変数を認識する
・関数のどの行を解析開始位置にするかを入力する
・解析開始位置に存在する変数のスライスを求める

＜機能分割部＞　解析結果ファイルを読み込んで分割結果ファイルを生成する．そのためにファイルを読み込んで機能を定義し，機能間で共通要素があればその部分を新たな機能として定義する処理を行う．

＜視覚化部＞　分割結果ファイルと中間ファイルを読み込み，疑わしい変数を入力されると，重みのあるコードに色を付けて画面に出力するために，以下の３つの処理を行う．

・分割結果ファイルを読み込んで機能を認識する
・マウスから入力された変数を基に重み付けを行う
・ソースプログラムの内容に色を付けて出力する

4.3　支援ツールの実現

支援ツールをWindows環境で使用することを目的に作成した．開発環境として，OSがMicrosoft Windows98のGateway社製のパソコン(GP7-500：CPUにPentium II 500 MHz，メインメモリに128 MB)を使用して開発した．各処理部は視覚化部をInprise社のBorland Delphi4で，その他はMicrosoft社のVisual C++6.0で作成した．各処理部のプログラム規模を表１に示す．

表１　各処理部のプログラム規模

名前	プログラム名	モジュール数	行数
前処理部	division.exe	7	747
解析部	slice.exe	34	2366
機能分割部	DivFunction.exe	9	430
視覚化部	FuncViewPro.exe	11	454

4.4　支援ツールの実行例

解析開始位置から各変数のスライスを求めて，機能分割部で機能分割を行うと，視覚部に処理が移る．ウインドウ中央下にあるselectと表示されているボタンをクリックすると，図５の変数選択ウインドウが開かれる．

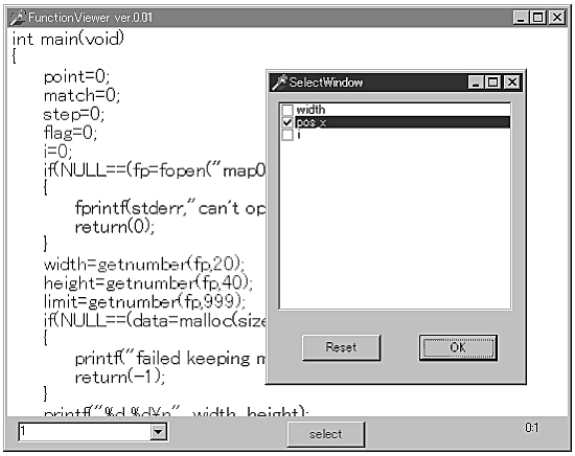

図５　視覚化部の出力例

変数選択ウィンドウには，解析開始位置に存在した変数が表示されているので，その中でバグの原因になっていると思われるものをチェックし，ＯＫボタンをクリックすると機能に重みが付けられ，変数選択ウィンドウが閉じられる．そして，メインウィンドウ左下のボックスで強調表示したい重みを選択すると，その重みをもつ機能の要素は赤色で表示される．

５．支援ツールの評価実験

この章では，支援ツールの性能を評価するために行った実験と実験結果について考察する．

5.1 評価実験の概要

＜実験環境＞

デバッグ実験は，本校の情報工学科棟３階の電子素子実験室を使用し，実験室にはGP7-500があり，それを用いて被験者にデバッグを行ってもらった．実験室には，デバッグ中の被験者の行動を記録するためにSony社製の8mmビデオカメラ(CCD-SC65)を備え付けた．

＜被験者＞

実験に参加してもらった被験者は４人で，いずれも本校情報工学科の５年生である．被験者は１・２年の時にPascal言語を，３年の時にＣ言語を学んでおり，プログラム歴は少なくとも４年以上である．また，４人とも卒業研究でＣ言語を用いてプログラムを作成している．

＜実験手順＞

実験はバグを含む２つのプログラムを，１回目はツールを用いずに，２回目はツールを用いてデバッグする．一人当りの所要時間は３時間程度であった．

①被験者は一人ずつ実験室に入室し，実験の内容と実験手順について5分程度の説明を受ける．

②デバッグするプログラムの仕様書・プログラムリストを渡し，どのようなバグが発生しているかを実演し，正常な場合の動作を説明する．

③仕様書に目を通し，その内容が理解できたらデバッグを開始する．

④被験者は，デバッグ中に考えていること，どのような原因でバグが起こっているかを発話してもらい，試験官は，その内容がバグの原因と一致している時，実験開始からの時間（バグ原因を特定した時間）を記録する．

⑤バグ原因が特定されてから，そのバグが正確に除去されたときの時間を記録する．この作業を全てのバグが正確に除去されるまで行う．

⑥１回目の実験が終了したら，少し休憩を入れて２回目の実験に移る．

⑦２回目と１回目との違いは，スライシングと機能分割，およびデバッグ時に用いるツールの説明であり，その他は１回目と同じである．

⑧２回の実験が終了したら被験者には，ツールに関するアンケートに答えてもらう．

＜デバッグ対象プログラム＞

「倉庫番」というパズルゲームと，コマンドを入力するとそのコマンドに対応した描画を行う「対話型作図ツール」である．バグはプログラム作成中に実際に起きたバグで，それぞれ２つずつ埋め込んである．

5.2 実験結果

４人の被験者による実験結果を表２に示し，実験開始後からの絶対時間を分単位で表わす．

表で色が付いているマスは，ツールを用いてデバッグしていたことを表している．各プログラムで，ツール使用者のグループと未使用者のグループの総デバッグ時間を比較したところ，倉庫番ではツールを用いると，用いないときより19％デバッグ時間が短縮され，作図ツールでは逆に18％余分にデバッグ時間がかかっていた．

表２　実験結果（単位：分）

	倉庫番				対話型作図ツール			
	バグ１		バグ２		バグ１		バグ２	
被験者	原因	除去	原因	除去	原因	除去	原因	除去
A	32	41	22	28	75	99	101	123
B	13	17	19	24	11	15	23	43
C	17	33	35	37	34	48	69	89
D	28	34	36	40	73	96	112	136

実験後にアンケートをとった結果，質問の回答は表３のようになった．アンケート内容は以下の通りである．ツールの操作性は良かったですか（操作性），ツールを用いてプログラムは理解しやすくなりましたか（理解度），ツールは便利だと思いますか（利便性），画面は見やすかったですか（画面構成），またツールを使いたいと思いましたか（再利用）の５項目について，５段階評価（１が悪い，５が良い）で行った．

表３　アンケート結果

被験者	操作性	理解度	利便性	画面構成	再使用
A	3	2	3	4	3
B	1	4	3	4	3
C	2	2	2	2	3
D	2	1	1	3	1
平均	2	2.3	2.3	3.3	2.5

操作性においては，被験者はストレスを感じており，ツールの使用が敬遠されていたようであった．

画面構成においては，平均的な評価が得られたが，重みがある命令に付ける色や，一画面で表示できるプログラムの行数などを変えられない点に不満があった．

5.3 考察

＜ツールとしての評価＞

作成した支援ツールのスライスを求め機能を分割するまで実行時間は，解析開始位置に存在する変数の数や，制御命令のネスト数，そして解析する関数の長さに比例する．実験でデバッグしてもらったプログラムにおいて，その両者を満たす場合で実行時間を計ったところ3.1秒しかかからなかった．そして平均実行時間は1.2秒

であり，被験者のデバッグ作業に支障をきたしていなかった．

以上のことより，作成したツールは，視覚化されるまでの実行時間や結果の表示法において，それほど問題はないが，ユーザインターフェースの部分のできが悪いのでツールの総合評価は良い結果が得られなかった．

＜デバッグ法の評価＞

表４は具体的にどのバグに作用しているかを示したものであり，実験開始後またはバグが除去されてからの相対時間である．

表４　各バグの短縮時間

	倉庫番		対話型作図ツール	
	１つ目	２つ目	バグ１	バグ２
ツールあり	25.5	6.5	73.3	32.6
ツールなし	30.7	8.5	55.9	33.6
短縮率	0.83	0.76	1.31	0.97

倉庫番の方は，１つ目・２つ目に除去されたバグのデバッグに要した時間を比較している．これは被験者Ａだけが，バグ２・バグ１と他の３人とは逆順で除去したためである．つまり，１つ目のバグは17%，２つ目のバグは25%デバッグ時間が短縮され，どちらのバグにもツールが有効に作用していた．

作図ツールの方は，バグ1は31%余分にかかり，バグ2は3%だけ短縮されており，ツールがバグ1には逆にデバッグの妨げとなっていた．

表５は，各バグに対して，どの命令を解析開始位置にしたときに読みとるコードが一番少なくなったかを示したものである．

表５　各バグの最小スライスサイズ

	倉庫番		対話型作図ツール	
	バグ１	バグ２	バグ１	バグ２
解析開始位置(行番号)	173	194	76	167
元の行数	11	11	58	25
スライスの行数	4	5	16	9

複数の変数に対してスライシングしていたが，平均３分の１程度まで短縮された．このことより正しく解析開始位置を設定すれば機能分割ができなくても，１つの変数に対するスライシングの時と同じくらいの早さでプログラムが理解できる．しかし，対話型作図ツールのバグ2のスライシングでは，元のプログラムが25行から9行までしか短縮されないのに対して，提案したデバッグ法は７行と少しであるがさらに短縮されていた．

また，実験を行って変数の数が多いほど制御命令に対する重みが高くなることがわかった．このことより提案したデバッグ法は，制御文の条件式設定ミスに対しては有効であると考えられる．

６．おわりに

本研究はデバッグにかかる時間を短縮するため，スライシング技術を利用したデバッグ支援ツールを作成した．その性能を評価することで，デバッグ法が有用であるかを考察した．ツールの性能を評価する実験を行った結果，一方のプログラムでは，ツールを用いるとデバッグ時間が19%短縮されたが，もう一方では18%余分に時間がかかる結果となった．

ツールは，プログラムの解析にかかる時間がほとんどなく，画面構成も普通という評価が得られたが，操作性が悪いため総合するとあまり良いツールではないと評価された．

デバッグ法は，今回の実験では，必要な処理が抜けているため発生したバグがほとんどだったため，提案した機能分割はあまり役に立たなかった．しかし，スライシングにより読みとるプログラムが平均３分の１程度まで減少でき，プログラム理解を支援できたと考えられる．

今後の課題としては，アンケートで指摘されていたツールの操作性を改善することや，構造体を含むプログラムを解析できない制約があるので，それに対応したアルゴリズムを実装することが挙げられる．

謝辞　日頃，ご指導を賜っている奈良先端科学技術大学院大学の鳥居宏次学長，松本健一教授及び門田暁人博士に感謝いたします．また，被験者として実験に協力をして頂いた学生諸氏にお礼を申し上げます．

参考文献

(1) Mark Weiser:"Program Slicing", IEEE Transactions on Software Engineering,Vol.SE-10, No.4" ,pp. 325-357 (1984).

(2) 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).

(3) 高田智規, 佐藤慎一, 井上克郎：“プログラム依存グラフの効率的な更新手法”, 電子情報通信学会論文誌Vol. J81-D-I, No.3, pp. 253-260 (1998).

(4) 西松顯, 地平稔, 楠本真二, 井上克郎：“関数呼び出し情報を用いたスライスサイズ削減のための一手法”, 電子情報通信学会論文誌Vol. J82-D-I, No.10, pp. 1256-1264 (1999).

(5) Francel M and Rugaber S:"The relationship of slicing and debugging to program understanding",Proceeding of 7th International Workshop on Program Comprehension,

pp.106-113(1999).

⑹ Lyle J and Weiser M":"Automatic Program Bug Location by Program Slicing", The Second International Conference on Computers and Applications, pp. 877-883 (1987).

⑺ 田中　洋：“関数呼び出しに対応した変数依存グラフに関する研究”, 奈良工業高等専門学校情報工学科, 平成9年度卒業研究報告書 (1998).

⑻ 内田眞司, 工藤英男 ,門田暁人, 松本健一, 井上克郎：“保守工程におけるバグ特定プロセスの分析”, 日本ソフトウェア科学会第16回大会論文集,pp.185-188 (1999).

⑼ 工藤英男, 内田眞司,門田暁人,松本健一：, “保守工程におけるデバッグ作業者のバグ特定プロセスの分析”,奈良工業高等専門学校, 研究紀要,第35号, pp. 75-80 (2000).

⑽ Shinji Uchida, Akito Monden, Hajimu Iida, Ken-ichi Matsumoto, Katsuro Inoue and Hideo Kudo： "Debugging process models based on changes in impressions of software modules,International Symposium on Future Software Technology 2000 (ISFST2000), pp. 57-62 (2000).

⑾ 中井伸郎：“スライシングを用いたデバッグ手法の研究”, 奈良工業高等専門学校情報工学科,平成12年度卒業研究報告書 (2001).