修士論文

題目

実行トレースの比較を用いたデバッグ手法の提案及び評価

指導教員

大野 和彦 講師

平成22年度

三重大学大学院 工学研究科 情報工学専攻
計算機アーキテクチャ研究室

松下 圭吾 (409M525)

内容梗概

我々は，デバッグの負担を軽減させる手法の研究を行っている．負担軽減の従来手法としてプログラムスライシングと呼ばれる手法がある．プログラムスライシングとは，全てのプログラムコードから注目すべき箇所を限定する手法である．しかし，この手法はバグと無関係なコードを多く抽出してしまうため，大幅な負担軽減は期待できない．

そこで本論文では，複数の実行トレースの比較により，「期待通りの結果となる入力パターンでの動作」と「期待値とは異なる結果となる入力パターンでの動作」の違いや共通部分の抽出を行い，スライシング結果からバグコードを推定する手法の提案を行う．性能評価の結果，プログラムスライシングのみを用いた場合と比較して，ソースコードの削減量が平均 15 %程度の向上を実現した．

Abstract

We research on reducing the burden of debugging. There is Program Slicing as a traditional technique of reducing user's burden. Program Slicing limits the program code to noteworthy code. However, because Program Slicing extracts the bugs and many extraneous codes to the bugs, it can not reduce a significant burden.

In this paper, we propose a method what narrows results of Program Slicing to related small part to the bugs by comparing execution traces. It extracts same or different parts between correct behavior and wrong behavior, and then estimates bug code. Results of performance evaluation, the amount of the extracted code by proposed method is about 15 percent fewer than the one of Program Slicing.

目 次

図 目 次

表 目 次

1　はじめに

近年，計算機科学の進歩によりソフトウェアは大規模かつ複雑になっている．それに伴い，デバッグを行うソフトウェア開発者の負担も増加してきている．プログラム作成において，テスト・デバッグ行程は作業量全体の 7 割を占めるとも言われており [1]，この行程の負担軽減は課題とされている．そのような理由から，様々なデバッグ手法 [2, 3, 4, 7, 8, 12] が研究されている．

負担軽減の方法として有力な方法は，作業の自動化である．設計書との矛盾点や変数値の調査等の作業を自動化出来れば，ユーザがデバッグを行う負担を軽減できる．しかし，自動的にデバッグを行う従来手法 [2, 3] は，対象プログラム論理プログラムと想定している事が多い．論理プログラムには制約充足が存在し，バグが存在すると制約に矛盾が発生する．従来手法は，これを利用して自動でバグの箇所を絞り込むのだが，命令型プログラムには適用できない．

命令型プログラムを対象とした研究の一つに，ソースコード全体からユーザが欲しい情報に関連したコードのみを抽出する手法 [4, 7, 8, 12] がある．本来，この手法の使用目的は，デバッグのみではなくコードの部品化やコード理解の補助も含んでいる．そのため，バグには関連のないコード片が含まれてしまう可能性が高く，あまり絞り込めない恐れがあるため，大幅な負担軽減は期待できない．

このような理由から，我々はデバッグの負担軽減に特化した自動ソースコード絞り込み手法の確立を研究の目的とした．バグは，その種類や存在箇所によってプログラムの動作や結果に様々な影響を与える．そのため，それら全てを対象とした手法を確立する事は非常に困難である．そこでまず，分岐命令に関するバグに着目した．分岐命令に関するバグは，変数の値だけでなく実行経路にも影響を与える．実行経路の誤りを手動で追う事は，ユーザにとって負担が大きい．

そこで本論文では，実行トレースの比較を用いた分岐命令に関するバグについてのデバッグ手法の提案する．性能評価を行った結果，プログラムスライシングのみを用いた場合と比較して，ソースコードの削減量が平均 15 %程度の向上を実現した．

以下，2 章では関連研究について述べる．3 章では提案手法の概要について述べ，4 章で具体的な提案手法の説明を行う．5 章で提案手法の実装方法について述べ，6 章で提案手法の評価結果を示す．そして，最後に 7 章でまとめを行う．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int output,input;
6    int point[5];
7
8    point[0] = 0;
9    point[1] = 1;
10   point[2] = 2;
11   point[3] = 3;
12   point[4] = 4;
13
14   scanf("%d",&input);
15
16   if(input > 4 || input < 0)
17     output = 0;
18   else
19     output = point[input];
20   printf("%d\n",output);
21   return 0;
22 }
```

図 2.1: c 言語プログラムの例

2 関連研究

本章では提案手法で用いたプログラムスライシングについて述べ，我々の目的を達成するために不足な点を明らかにする．

2.1 プログラムスライシング

プログラムスライシングとは，全てのプログラムコードから注目すべき箇所を限定する方法として Weiser が提案した手法 [4] である．注目すべき箇所とは，ユーザが知りたい情報に直接的または間接的に関係する箇所の事である．この抽出された箇所の事をスライスと呼ぶ．実験から，実際のデバッグに対して効果があることを示した論文 [5] もある．スライシングは，その性質により，静的スライシングと動的スライシングに分けられる．以下では，それぞれのスライス抽出過程と特徴について述べる．

2.1.1 静的スライシング（SPS:Static Program Slicing）

SPS は，静的にソースコードを解析し，依存関係を抽出する手法である．ソースコードに含まれる文間の依存関係を抽出する事ができる．以下に依存関係の定義を示す．

ソースプログラム p 中の文 s_1 と s_2 について考える．

- 以下の条件を満たす時，文 s_2 は s_1 に依存しており，その関係を制御依存 (CD:Control Dependence) 関係という．
  1. s_1 が制御文である
  2. s_1 の結果により，s_2 が実行されるかどうか決定される
- 以下の条件を満たす時，文 s_2 は s_1 に依存しており，その関係をデータ依存 (DD:Data Dependence) 関係という．
  1. s_1 において，変数 v が定義される
  2. s_1 から s_2 において，v を再定義しない経路が少なくとも一つは存在する
  3. s_2 において，v が参照される

この依存関係を用いて，プログラム依存グラフ (PDG:Program Dependence Graph)[6] を作成する．PDG は辺が文間の依存関係を表し，節点が制御文・代入文などの文を表す有向グラフである．有向辺の向きは，プログラムの流れを把握しやすくするために，依存の向きとは逆向きである．

文 s における変数 v に関する静的スライスとは，文 s に対応する PDG 節点から PDG 辺を逆向きに辿ることで到達可能な節点集合に対応する文の集合である．制御依存辺は，無条件に辿る事が出来る．データ依存辺は，最初の節点の場合は着目する変数 v に対応する辺のみ辿る事が可能で，後は無条件で辿る事が出来る．なお，ユーザが着目している文と変数の組 (s, v) を静的スライシング基準 (Static Slice Criteria) と呼ぶ．

図 2.1 のプログラムに対して，静的スライシング基準 $(20, output)$ でスライシングを行った結果を図 2.2 に示し，そこから作成される PDG を図 2.3 に示す．図 2.3 は，ノードが各行，点線が CD, 実線が DD を表している．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5   int output,input;
6   int point[5];
7
8    point[0] = 0;
9    point[1] = 1;
10  point[2] = 2;
11  point[3] = 3;
12  point[4] = 4;
13
14  scanf("%d",&input);
15
16 if(input > 4 || input < 0)
17    output = 0;
18  else
19    output = point[input];
20  printf("%d\n",output);
21  return 0;
22 }
```

図 2.2: 静的スライシングの結果　(20,output)

2.1.2　動的スライシング（DPS:Dynamic Program Slicing）

DPS とは，Agrawal らが提案した手法 [7] である．SPS ではソースコードを対象に依存関係を解析し，スライスを抽出するが，DPS の解析対象は実行系列になる．実行系列とは，ある入力によりプログラムを実行した時の，実行された文の列である．実行系列中の r 番目の文の実行のことを実行時点 r と呼ぶ．

実行系列 e 中の実行時点 r_1 と r_2 について考える．

- 以下の条件を満たす時，実行時点 r_2 は r_1 に依存しており，その関係を動的制御依存 (DCD:Dynamic Control Dependence) 関係という．
  1. r_1 が制御文である
  2. r_1 の結果によって，r_2 が実行されるかどうか決定される
- 以下の条件を満たす時，実行時点 r_2 は r_1 に依存しており，その関係を動的データ依存 (DDD:Dynamic Data Dependence) 関係という．

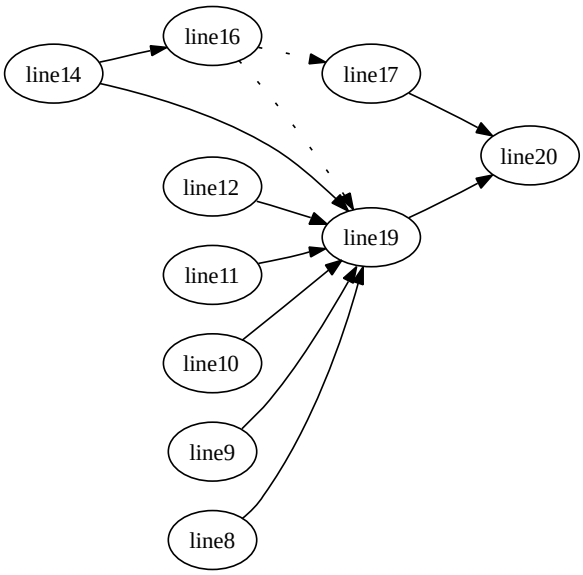

図 2.3: プログラム依存グラフ

1. r_1 において，変数 v が定義される
2. r_1 から r_2 において，v を再定義する実行時点がない
3. r_2 において，v が参照される

この依存関係を用いて，動的依存グラフ (DDG:Dynamic Dependence Graph) を作成する．PDG と同様に有向グラフであり，辺は依存関係を表している．PDG との違いは，節点が実行時点を示している点である．

入力値の組 X を与えた時の実行時点 r における変数 v に関する動的スライスとは，実行時点 r に対応する DDG 節点から DDG 辺を逆向きに辿ることで到達可能な節点集合に対応する文の集合である．制御依存辺は，無条件に辿る事が出来る．データ依存辺は，最初の節点の場合は着目する変数 v に対応する辺のみ辿る事が可能で，後は無条件で辿る事が出来る．なお，入力値の組とユーザが着目している文と変数の組 (X, s, v) を動的スライシング基準 (Dynamic Slice Criteria) と呼ぶ．

図 2.1 のプログラムに対して，動的スライシング基準 $(\{3\}, 20, output)$ でスライシングを行った結果を図 2.4 に示し，そこから作成される DDG を図 2.5 に示す．図 2.5 は，ノードが各行，点線が DCD，実線が DDD を表している．

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int output,input;
6    int point[5];
7
8    point[0] = 0;
9    point[1] = 1;
10   point[2] = 2;
11   point[3] = 3;
12   point[4] = 4;
13
14   scanf("%d",&input);
15
16  if(input > 4 || input < 0)
17    output = 0;
18  else
19    output = point[input];
20  printf("%d\n",output);
21  return 0;
22 }
```

図 2.4: 動的スライシングの結果　({3},20,output)

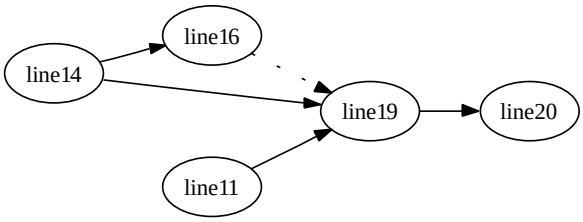

図 2.5: プログラムの動的依存グラフ

2.1.3　不足している点

プログラムスライシングの目的は，我々の目的と類似しており，対象プログラムが命令型である点も同じである．しかし，依存関係のあるコードが全て抽出されてしまうため，正しく記述されているコードが多く含まれてしまう．

SPS では，実行に関係のないコードが含まれてしまう．また，ある入

力の実行において関係のないコードを省くDPSでさえも，依存関係が複雑な場合，多くのコードが抽出されてしまう．

そこで我々は不足点を補うために，実行トレースの比較によって，スライシングで得られた結果からバグを含む小範囲を切り出す．

3　提案手法の概要

デバッグは，入力パターン（入力として取り得る値の集合）による動作毎にステップ実行を繰り返しながら変数の値を確認するのが一般的である．

この方法の問題点は，2つ存在する．1つ目は，ユーザに課せられる負担（依存関係の把握，作業内容の記憶等）が大きい点である．2つ目は，バグの影響を受けているソースコード行が判断しにくい点である．

これらの問題点を解決するためには,「絞り込み」と「自動化」が必要である.「絞り込み」を行うためには，あるコードがバグであると推定できるだけの情報が必要となる．そこで複数の入力で実行し,「期待通りの結果となる入力パターンでの動作」（以下，CR）と「期待値とは異なる結果となる入力パターンでの動作」（以下，WR）の違いや WR 同士での共通部分の抽出を行う．

WR の内 CR と共通している部分は，CR では期待した結果となる事を考慮すると，その部分にバグが存在する可能性は低いと考えられる．特定の入力でのみバグの影響を受けるとすれば，他の入力とは異なるその入力特有の動作，つまり CR と WR の差分の一部にバグが存在する可能性が高い．また，WR の中で共通部分があれば，どの WR もその共通部分でバグの影響を受けている可能性がある．

「動作」とは抽象的な言葉であり，具体的な情報は複数存在する．プログラム実行途中のある地点における変数値や実行経路等も「動作」情報に含まれる．全ての「動作」情報を取得するとなると，膨大な量のデータを得る事になり，人が手動で扱う事が難しい．これを解決するため,「動作」情報の中から必要な情報を選びだし，得られた情報の比較・統計を「自動」で行う．その結果から，バグが存在する箇所を推定し，ユーザに表示する事で負担軽減を行う．

この方法は，絞り込んだ結果にバグが存在しない可能性もある．もし，ユーザと何度も質疑応答を繰り返しながら絞り込みを行った結果にバグが存在しなければ，そのユーザに掛かる負担はより大きくなる．しかし，自動で絞り込みを行うのであれば，結果が出るまで動作を監視する必要がない．その結果にバグが存在すれば探索に掛かる負担が軽減され，バグが存在しない場合にも提案手法を用いない場合のユーザに掛かる負担とあまり変わらない．

4　実行トレースの比較を用いたデバッグ手法

本章では，前章で述べた概要を具体化した手法の説明を行う．

4.1　対象とするバグ

様々なバグが存在する中で，我々が着目したのは「分岐命令の条件式」に含まれるバグである．このバグは，期待と異なる実行経路を辿る事態を引き起こすが，実際のバグの原因としては不等号が逆向きに記述されている等の簡単な誤りである事が多い．しかし，プログラム規模が大きくなるにつれて，ユーザがこの誤りを手動で探しだすことは困難となる．そのため，このバグを自動で検出できれば，ユーザの負担軽減につながる．

4.2　対象バグの影響

「分岐命令の条件式」に含まれるバグというのは，その分岐命令に制御依存関係を持つコード行に対して影響を与える．そのため，バグの影響によってあらゆる入力において実行されないコード行が発生する可能性がある．ある入力において実行されないコード行というのは，現実的に数多く存在する．しかし，入力パターンが増える程，全ての入力で実行されないコード行はバグの影響を受けている可能性が高くなる．

また，ある行の実行回数が入力パターンに対して相関関係を持つ事がある．全ての入力において WR となるのであれば，その相関関係も正しいとは言い難い．しかし，連続した入力（連続した自然数等）において，ある境界点で CR と WR が分かれ，CR 側では相関関係があり，WR 側では相関関係が崩れている場合，またはその逆の場合，その行はバグの影響を受けている可能性が高い．

4.3　入力パターンの決定

複数の入力により実行する場合，与えられた入力によって得られるデータの性質は大きく異なる．例えば，複数の乱数を入力として実行した場合，入力値の偏りを無くす事ができる．しかし，入力との間の相関関係の乱れを正確に読み取ることが出来ない．

このような関係性は，入力値が数値でない場合にも存在する．例えば，文字数や頭文字等に相関関係を持つこともある．このように様々な入力が想定されるが，今回は一番単純なケースとして，実行時引数が自然数１つであるプログラムを対象とする．

この理由は，引数が複数の場合，引数毎に関係性を調査すれば引数が１個の時と同様の処理を行え，文字である場合，文字列を数値に変換（文字数等）出来れば同様の処理を行えると考えたからである．効率化や信頼性向上を行うためには，適切な範囲の調査や適度な間引きが必要になるが，今回は言及しない．

4.4 実行トレース比較方法

まず複数の入力パターンにおける実行トレースを取得する．提案手法では，実行された行番号と各行が実行された回数の 2 つの情報を取得する．それらを用いて，ソースコードの各行について以下の処理を行う．

1. 与えられた入力パターンにおいて実行回数が 0 であるかを判定
2. 各入力間における実行回数の差分を算出
3. 実行回数の差分が常に 0 であるかを判定
4. 各実行回数差分間における差分を算出
5. 実行回数差分間の差分が常に 0 であるかを判定
6. 実行回数差分間の差分が 0 以外で一定であるかを判定

（1）は，その行が少なくとも 1 つ以上の入力で実行されるかどうかを判定している．もしこの判定が真であれば，その行は与えられた入力パターンでは到達出来ない行である事が分かる．

（2）,（3）では，その行が入力に関係なく実行回数が一定回数であるかを判定している．もしこの判定が偽となる場合，その行はある境界から実行回数がそれまでの入力における実行回数とは異なり，それまでの傾向とは異なる性質を示す事になる．

（4）,（5）,（6）では，その行の実行回数の増減量が一定であるかを判定している．増減量が 0 で一定であれば，この行の実行回数は等差的に変化している事になる．これも一定回数の判定と同様で判定が偽となる場合，その入力はそれまでの傾向とは異なる性質を示す事になる．

このようにして，各行の性質を決定する．今回使用した性質の種類を以下に示す．

***BLANK OR NONEXEC**
空行，一度も実行されない，gdb のステップ実行で止まらない

***SAME FREQUENCY**
どの入力においても実行回数が一定である

***SAME FRE INJURED**
ある入力までは SAME FREQUENCY を満たすが，それ以降は異なる

***EQUAL DIFFERENCE**
入力に応じて等差的に実行回数が変化している

***EQUAL DIF INJURED**
ある入力までは EQUAL DIFFERENCE を満たすが，それ以降は異なる

***EQUAL DIFDIF**
実行回数の差分が一定の増減を行いながら変化している

***EQUAL DIFDIF INJURED**
ある入力までは EQUAL DIFDIF を満たすが，それ以降は異なる

***NOT DEFINED**
上記以外の行

4.5 簡易版動的スライシング

簡易版動的スライシング [8] は，SPS から実行トレースを用いて未実行の行を削除する事で，DPS に近い結果を得ることができる．DPS の利点は，データ依存関係の詳細な情報（例えば，何番目の配列要素に依存しているのか等）を取得できる点である．しかし，DPS は動的に依存解析を行うため，処理量が多くなり実行時間が長くなってしまう．一方，簡易版動的スライシングは，静的解析と最低限の動的情報を利用するので，処理量が少ない．そこで提案手法では，実行トレースを取得が利用できる簡易版動的スライシングを利用する．

4.6　バグの箇所を絞り込む方法

バグの箇所を実行トレース・分岐命令ブロックの箇所・簡易版動的スライシングを用いて絞り込む.

絞り込みの手順を以下に示す.

1. ユーザが指定した明らかにバグの影響を受けているコードをスライス基準として簡易版動的スライシングを行う
2. スライスに含まれる分岐命令ブロックを検出
3. ブロック内の各行の性質を調査
4. ブロック内の全ての行が BLANK OR NONEXEC であればバグとし，条件式の行を表示
5. バグとするブロックが 1 つもない場合，スライスに含まれる行で性質が SAME FRE INJURED，EQUAL DIF INJURED，EQUAL DIFDIF INJURED である行を表示
6. 対象とするコード行が無ければ，スライスを全て表示

これらの処理により，対象とするコード行が存在している場合は，小範囲を切り出すこと出来る．対象とするコード行が無い場合は，簡易版動的スライシングの結果となる．以上の手法を用いることで，絞り込み結果にバグが存在しない可能性を少なくしている.

4.7　バグプログラムに対する使用例

図 4.6 のプログラムを用いて提案手法の使用方法を述べる．このプログラムに存在するバグは，29 行目の条件式の不等号が逆になっている事である．このバグを提案手法により抽出する.

```
1  #include <stdio.h>
2  #define N 10
3
4  int main(int argc,char **argv){
5      int i=0, j = 0, x=0, y=0;
6      int n = 0;
7      int d[N][N];
8
9      n = atoi(argv[1]);
10
11     for(i=0;i < n;i++)
12      for(x=0;x < n;x++)
13         {
14            d[i][x] = 0;
15         }
16
17
18    for (i = 1 ; i < n ; i++)
19      {
20         d[i][0] = 1;
21          j=1;
22         while (j <= i - 1)
23            {
24               d[i][j] = d[i-1][j-1] + d[i-1][j];
25                j ++;
26            }
27     }
28
29    for (y = 0; y > n; y++)
30       {
31          for (x = 0; x < n-y; x++)
32              printf("  ");
33          for (x = 0; x < y; x++)
34              printf("%3d ", d[y][x]);
35          printf("\n");
36       }
37    return 0;
38 }
```

図 4.6: バグが存在するプログラム

まず，このプログラムの実行トレースを取得する．与えた入力パターンは，自然数の 1〜10 である．その結果，各行の実行回数とその結果によっ

て決められた性質が表 4.1 となる．1 行目の数値 x は，入力値を示している．表 4.1 に記述されていない行番号は，全て BLANK OR NONEXEC である．

表 4.1: 各行の各入力値における実行回数と各行の性質

行番号	入力値 1	2	3	4	5	6	7	8	9	10	性質
5	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY
6	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY
9	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY
11	2	3	4	5	6	7	8	9	10	11	EQUAL DIFFERENCE
12	2	6	12	20	30	42	56	72	90	110	EQUAL DIFDIF
14	1	4	9	16	25	36	49	64	81	100	EQUAL DIFDIF
18	1	2	3	4	5	6	7	8	9	10	EQUAL DIFFERENCE
20	0	1	2	3	4	5	6	7	8	9	EQUAL DIFFERENCE
21	0	1	2	3	4	5	6	7	8	9	EQUAL DIFFERENCE
22	0	1	3	6	10	15	21	28	36	45	EQUAL DIFDIF
24	0	0	1	3	6	10	15	21	28	36	EQUAL DIFDIF
25	0	0	1	3	6	10	15	21	28	36	EQUAL DIFDIF
29	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY
30	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
31	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
32	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
33	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
34	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
35	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
36	0	0	0	0	0	0	0	0	0	0	BLANK OR NONEXEC
37	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY
38	1	1	1	1	1	1	1	1	1	1	SAME FREQUENCY

次に，分岐命令ブロックに含まれる各行の性質を調べる．このプログラムに含まれる分岐命令ブロックは，[11-15][12-15][18-27][22-26][29-36][31,32][33,34] の 7 つが存在する．ここで，34 行目をスライス基準として簡易版動的スライスを行うと，[31,32][33,34] の 2 つが今回の実行には関係がないブロッ

クであると判断され，バグ候補から外れる．残りのブロックに含まれる行を表 4.1 と照らし合わせると，[29-36] は先頭の行以外は全て BLANK OR NONEXEC となっている事が分かる．そこで，[29-36] の先頭の行である 29 行目をバグが存在する行であるとみなし，ユーザに表示する．このように，提案手法を用いる事でソースコード全体からバグが存在する行を自動で特定する事が出来る．

5　提案手法の実装方法

評価に用いた提案手法の実装方法を述べる．現在は実験段階であり，統合されたツールとして完成していない．各部分を個別に作成し，それらの出力をファイルに保存し，次の処理の入力としている．全体の処理の流れを図5.7に示す．Toolの数字は，節と対応している．以下，特に記述の無い部分はC言語で実装している．

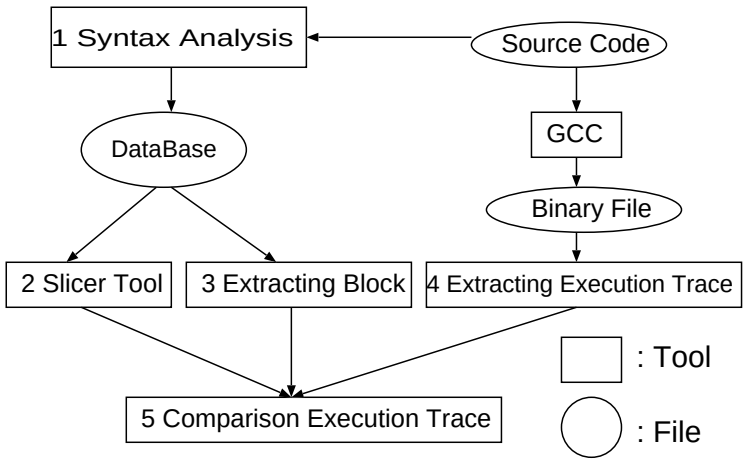

図 5.7: 提案手法のフロー図

5.1　構文解析

構文解析にはSapid[9]を使用した．Sapidは，オープンソースのCASEツールである．対象としている言語は，C言語やJavaである．Sapidは，「sdb4」コマンドを使用することで，引数で指定されたソースコードの静的解析を行い，構文解析を行う．それらの結果をデータベース（DB）として保管し，専用のアクセスルーチン（AR）を用いる事で，必要な情報を得ることができる．

5.2　スライサーツール

スライサーツールは，SapidとXPathを用いて作成した．まず，Sapidで用意されている「mkFlowView」コマンドを実行し，DBにプログラム

内の依存関係を作成する．次に,「spdMkCXModel」コマンドを実行し，先程作成した依存情報を XML 形式でファイルに出力する．

その後，作成したファイルに対して XPath を用いて必要な情報を取得する．XPath[10] とは，XML ファイルを扱うための Perl 用のモジュールであり，XML ファイル内の特定の要素情報を抜き出す事が容易に行える．今回は，XML ファイルから依存関係が記述されている箇所を取得するために用いている．

その次に，XPath によって得られたデータに対して文字列分割を行い，依存関係の情報を取得する．それらの情報をもとに，各行間の依存関係を構造体に保存する．構造体のメンバは，[行番号] と [この行に依存している行番号へのポインタ配列] と [この行が依存している行番号へのポインタ配列] としている．これにより，任意の行番号から依存関係を辿る事が可能となる．

5.3 分岐命令ブロック抽出

分岐命令ブロック抽出にも，Sapid と XPath を利用している．スライサーツールで作成した XML ファイルに対して，XPath を用いて分岐命令ブロックの情報が記述されている箇所を抜き出す．そこから，文字列分割を行い，ブロックの行番号を取得する．

5.4 実行トレース取得

実行トレースの取得には GDB を用いる．提案手法で用いる情報は，各行の実行回数である．そこで，プログラムをステップ実行の繰り返しにより，終了まで実行する．その実行中に得られる GDB のログを保存し，実行終了後に集計を行った．ステップ実行のログは，次に実行される [行番号] と [命令] なので我々が必要としている情報と一致している．

5.5 実行トレース比較

各実行トレースを入力とし，各行番号毎に処理を行う．まず，処理を行う行番号のデータを各実行トレースから取得し，配列に保存する．次に，その配列に対して 4.4 節で述べた処理を行う．その結果をもとに，各行番号が持つ性質を決める．

その後，スライサーツールの結果と分岐命令ブロック抽出の結果を読み込み，4.6 節の処理を行う．

6　性能評価

6.1　評価方法

評価に用いたプログラムを以下に示す.

***Pascal**　入力された段数のパスカルの三角形を
表示するプログラム

***Monte**　入力された試行回数のモンテカルロ法で
円周率を求めるプログラム

***GA**　入力は街の数であり，それらの街を全て巡回する最適経路を遺伝子アルゴリズム（GA）で
導き出すプログラム

これらのプログラムに対して，ランダムな位置に指定した種類のバグを挿入するツールを使用し，分岐命令の条件式に関するバグを 1 つずつ挿入したプログラムを作成した．作成したそれぞれのバグの内容を表 6.2 に示す.

表 6.2: 作成されたバグ

プログラム名	変更内容
Pascal	″ <″⇒ “ >″
Monte	″ <=″⇒ “ >=″
GA1	″ <″⇒ “ >″
GA2	″ <″⇒ “ >″
GA3	″ <″⇒ “ >″
GA4	″ ==″⇒ “! =″
GA5	″ <=″⇒ “ <″

6.2　評価結果

表 6.2 に示したバグが存在するプログラムに対して，提案手法を用いた結果を表 6.3 に示す．DPS と提案手法は抽出された行数を示しており，削

減率とはソースコード全体から何%削減したかを示している．与えた入力パターンは，自然数 1〜10 である．

表 6.3: 分岐命令の条件式バグ　評価結果

プログラム	総行数	DPS	削減率	提案手法	削減率
Pascal	38	15	61(%)	1	97(%)
Monte	35	9	74	1	97
GA1	431	70	84	1	99
GA2	431	67	84	1	99
GA3	431	69	84	1	99
GA4	431	73	83	73	83
GA5	431	73	83	73	83

DPS の結果は，平均すると約 79 %の削減率を示している．一方，提案手法の結果は，平均で約 94 %の削減率を示した．従って，DPS と比べて削減率が平均 15 %程度の向上を示している事がわかる．

6.3　結果考察

評価結果として，DPS と比べて，ソースコード量の削減率が平均 15 %程度の向上がみられた．もし，条件式で参照される変数値の算出にバグが存在する場合，提案手法で抽出された行にはバグが無い事になる．しかし，提案手法の結果である 1 行を調査する負担は小さく，その後の調査はその行が実行されるより前に実行された部分の調査になるため，それ以降の部分の調査を行う負担を省くことができる．

それに加えて，対象とするバグが存在しない場合に関しても，DPS に近い結果となるのでバグを取りこぼす可能性はほとんどない．この手法で取りこぼすバグは，配列領域外アクセス等によるデータ破壊である．これは依存関係のないコードから影響を受ける可能性があるので，スライシングでは発見する事が出来ない．このようなバグに対応するためには，メモリ管理が必要となる．

7　おわりに

今回，我々は実行トレースの比較を用いたデバッグ手法を提案した．評価の結果，DPS と比べて平均 15 %程度の削減率の向上がみられた．この結果により，DPS よりも提案手法はデバッグに有効であるといえる．

今後の展望としては，静的解析を詳細に行う事で各ループの実行傾向を事前に把握し，実行トレースと照らし合わせる事でより正確な絞り込みを可能にする．現時点では，明らかに異常な実行回数であるバグ以外は推定できない．静的解析によって各ループの特徴（定数回ループ，ループ回数が入力値に比例等）を検出できれば，その特徴とは異なる値を示したループをバグとして検出する事ができる．このようにしてバグの箇所を推定する精度を向上させ，提案手法をより実用的にしていく．

謝辞

本研究を行うにあたり，御指導，御助言頂きました大野和彦講師，並びに多くの助言を頂きました近藤利夫教授，佐々木敬泰準教授に深く感謝致します．また，様々な局面にてお世話になりました研究室の皆様にも心より感謝いたします．

参考文献

[1] 2008 年版組み込みソフトウェア
産業実態調査報告書
http://www.meti.go.jp/policy/mono_info_service
/joho/2008software_research.html

[2] Reiter,R.:”A Theory of Diagnosis from First Principles”,*Artificial Intelligence*, Vol.32,pp.57-95(1987).

[3] 網代育大, 長健太, 上田和紀: ”静的解析と制約充足によるプログラム自動デバッグ ”, コンピュータソフトウェア,Vol.15,No.1,pp.54-58(1998).

[4] M.Weiser:“Program Slicing”, IEEE Transaction on Software Engineering, 10(4),pp.352-357(1984).

[5] 西松顯, 楠本真二, 井上克郎:フォールト位置特定におけるプログラムスライスの実験的評価 ”, 信学技報,SS98-3, May(1998).

[6] K.J.Ottenstein and L.M.Ottenstein:”The program dependence graph in a software development environment”,Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Enviroments, pp.177-184(1984).

[7] H.Agrawal and J.Horgan:“Dynamic Program Slicing”, SIGPLAN Notices,Vol.25,No.6,pp.246-256(1990).

[8] Gupita, R., Soffa, M.L. and Howard, J.:”Hybrid Slicing: Integrating Dynamic Information with Static Analysis”,*ACM Transaction on Software Engineering and Methodology*, Vol.6,No.4,pp.370-397(1997).

[9] Sapid
http://www.sapid.org/index-ja.html

[10] XPath
http://docstore.mik.ua/orelly/xml/pxml/ch08_02.htm

[11] Bates, S. and Horwitz, S.: "Incremental Program Testing Using Program Dependence Graphs",*Conference Record of the Twentieth ACM Symposium on principles of Programming Languages*, 1993.

[12] 佐藤慎一, 飯田元, 井上克郎: "プログラムの依存関係解析に基づくデバッグ支援システムの試作 ", 信学技報,SS95-4,May(1995).