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

松下圭吾[†] 松本真樹[†] 大野和彦[†]
佐々木敬泰[†] 近藤利夫[†] 中島浩[††]

デバッグの負担を軽減する方法の一つに，バグを含むと推測されるコード範囲を絞り込む手法がある．プログラムスライシングはプログラムコード全体から注目すべき箇所を限定する手法であり，このようなデバッグ支援に利用できる．しかし，プログラムスライシング単独ではバグと無関係なコードも多く抽出されるため，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の存在箇所を推定する手法を提案する．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 %程度向上した．

A Debugging Method Based on Comparison of Execution Trace

Keigo Matsushita,[†] Masaki Matsumoto,[†] Kazuhiko Ohno,[†]
Takahiro Sasaki,[†] Toshio Kondo[†] and Hiroshi Nakashima[††]

Filtering program code to obtain code fragments with bugs is one of the automated debugging methods. *Program slicing* extracts code fragments concerning the focused variable and can be used for such debugging. However, the result of slicing contains many lines without any bugs and not satisfying for such purpose.

Therefore, we propose a new debugging method. Our method compares multiple execution traces and extracts the common and uncommon parts between the cases whose behaviors are *correct* and *wrong*. Using such information, the amount of code fragments obtained by slicing can be reduced. The evaluation shows our method reduced the slicing result 15% in average.

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)~8)] が研究されている．

負担軽減の方法として，仕様との整合性チェックやバグの存在箇所を絞り込むといった手法が提案されている．前者は自動デバッグが期待できる反面，需要の高い手続き型言語への適用が難しい．一方，後者に利用できる技術として，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プログラムスライシング[5)~8)] が研究されているが，デバッグ以外にコードの部品化やコード理解の補助といった用途も目的としており，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い．このため，そのままではバグの絞り込み手段として十分な効果が得られないことがある．

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を研究している．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．本論文では，分岐命令に関するバグを対象として，実行トレースの比較を用いたデバッグ手法を提案する．

以下，2 章では関連研究について述べる．3 章では

† 三重大学 大学院工学研究科
Graduate School of Engineering, Mie University
†† 京都大学 学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University

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．

2. 関 連 研 究

2.1 自動デバッグ手法

これまでに様々な自動デバッグ手法が提案されており，仕様との整合性をチェックする，プログラム上の非整合性やエラーパターンをチェックする，正しい箇所を除外することでバグを含む箇所を絞り込む，といったアプローチがとられている[2]．

論理型・関数型言語は制約充足などを用いた形式的検証に適しており，言語上，あるいは仕様との整合性をチェックする自動デバッグ手法が提案されてきている[3),4]．しかし，このような手法を手続き型言語に適用することは難しい．

これに対し，無関係なコードを除外することでバグを含む箇所を絞り込むアプローチは，言語を問わず適用可能である．必ずしもバグの位置が特定できるとは限らないが，絞り込みによりその後のデバッグの作業量が減るため，ユーザのデバッグを支援する手法としては実用性が高い．このため，本研究ではプログラムスライシングと呼ばれる手法を元に，バグの存在箇所を絞り込むアプローチを採る．

2.2 プログラムスライシング

プログラムスライシング[5] とは，プログラムコード全体から注目すべき箇所を抽出する手法である．ここで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であり，抽出結果をスライスと呼ぶ．プログラムスライシングは実際のデバッグに対して効果があることも報告されている[9]．

具体的な手法として，静的スライシングと動的スライシングがある．以下，それぞれのスライス抽出過程と特徴について述べる．

2.2.1 静的スライシング

静的スライシング（SPS: Static Program Slicing）では，静的にソースコードを解析し，文間の依存関係を抽出する．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，

- 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

```
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 1 C 言語プログラムの例
Fig. 1 Example of C Program

- 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）[10] 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表す．

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の集合に対応する，文の集合である．このとき，制御依存辺は無条件に辿る事が出来る．データ依存辺は，最初の節点からは着目する変数 v に対応する辺のみ，他の場合は無条件で辿る事が出来る．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 1 のプログラムに対して，静的スライシング基準 $(20, \texttt{output})$ でスライシングを行った結果を**図 2** に，そこから作成される PDG を**図 3** に，それぞれ示す．図 3 は，ノードが各行，点線が CD, 実線が DD を表している．

2.2.2 動的スライシング

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のに対し，動的スライシング（DPS: Dynamic Program Slicing）[6] では，実行系列を解析

```
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 静的スライシングの結果 (20, output)
Fig. 2 Static Program Slicing (20, output)

```
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 4 動的スライシングの結果 ({3}, 20, output)
Fig. 4 Dynamic Program Slicing ({3}, 20, output)

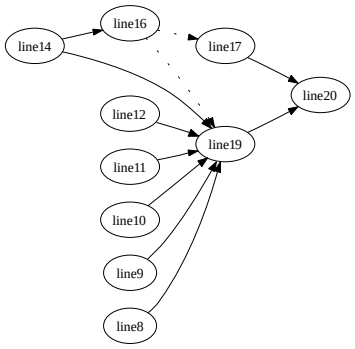

図 3 プログラム依存グラフ
Fig. 3 Program Dependence Graph

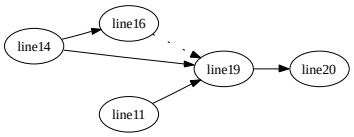

図 5 プログラムの動的依存グラフ
Fig. 5 Dynamic Dependence Graph

対象とする．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り，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，

- 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．
  (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．DDG は 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．このとき，制御依存辺は，無条件に辿る事が出来る．データ依存辺は，最初の節点からは着目する変数 v に対応する辺のみ，他の場合は無条件で辿る事が出来る．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 1 のプログラムに対し，動的スライシング基準 $(\{3\}, 20, \texttt{output})$ でスライシングを行った結果を**図 4** に，そこから作成される DDG を**図 5** に，それぞれ示す．図 5 は，ノードが各行，点線が DCD，実線が DDD を表している．

3. 提案手法の概要

現状のデバッグ手法としては，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．この方法には，依存関係の把握や作業内容の記憶などユーザに課せられる負担が大きい，バグの影響を受けているソースコード行がどこか判断しにくい，といった問題点がある．

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．そこで複数の入力に対して実行を行い，「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 のうち 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 の差分の中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か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負担とあまり変わらない．

2.2 節で述べたプログラムスライシングは，手続き型言語のプログラムを対象として自動的に絞り込みを行う点で，我々の目的に適している．しかし，スライシング基準から依存関係のあるコードを全て抽出するため，ある入力の実行に対し関係のないコードを省くことのできる DPS を用いても，依存関係が複雑な場合には広範囲のコードが抽出されてしまい，その中にはバグと無関係な正しいコードが多数含まれる．

そこで我々の手法では，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範囲を絞り込む．

4. 実行トレースの比較を用いたデバッグ手法

4.1 対象とするバグ

様々なバグが存在する中で，今回は「分岐命令の条件式」に含まれるバグを対象に手法の定式化を行った．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は現実的に数多く存在するが，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）に対しある境界点で 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，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し，例えば文字列であれば文字数や頭文字等に相関関係を持つことがある．今回は一番単純なケースとして，実行時引数が自然数 1 個であるプログラムを対象とする．引数が複数の場合でも引数毎に関係性を調査することで，引数が文字列の場合も文字数など何ら

かの数値に変換することで，いずれも引数が自然数 1 個の時と同様の処理を行える．処理の効率化や結果の信頼性向上には，調査範囲や間引き間隔を適切に設定する必要があるが，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[7] 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，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，簡易版動的スライシングを使用する．

4.6 バグ存在行の絞り込み

バグが存在する行を，実行トレース・分岐命令ブロック・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する．
(3) ブロック内の各行の性質を調査する．
(4) ブロック内の全ての行が BLANK OR NONEXEC であればバグとし，条件式の行を表示する．
(5) バグとされるブロックが 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 である行を表示する．
(6) 対象とするコード行が無ければ，スライスを全て表示する．

この手順により，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が出来る．対象とするコード行がない場合は，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 使 用 例

図 6 のプログラムを用いて，提案手法の使用方法を説明する．このプログラムに存在するバグは，29 行

表 1 各行の各入力値における実行回数と各行の性質
Table 1 Number of Execution and Type for Each Line

行番号	入力値										性質
	1	2	3	4	5	6	7	8	9	10	
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

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．

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 1〜10 である．各行の実行回数とその結果によって決められた性質を**表 1** に示す．示されていない行番号は，全て BLANK OR NONEXEC である．

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，[11-16], [12-15], [18-27], [22-26], [29-36], [31,32], [33,34] の 7 つである．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 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 7** に示す．Tool の番号は，節と対応している．以下，特に記述のない部分は C 言語で実装している．

5.1 構 文 解 析

構文解析には Sapid[11] を使用した．Sapid は，C や Java を対象とするオープンソースの CASE ツールである．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・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[12] 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抽出する．次に，`spdMkCXModel` コマンドを実行し，依存情報を XML 形式でファイルに出力する．

その後，このファイルから XPath を用いて依存関係情報を取得する．XPath は XML ファイルを扱うための Perl 用のモジュールであり，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．得られた情報は，各行間の依存関係として構造体に保存する．構造体のメンバは，[行番

```
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 6 バグが存在するプログラム
Fig. 6 A Program with a Bug

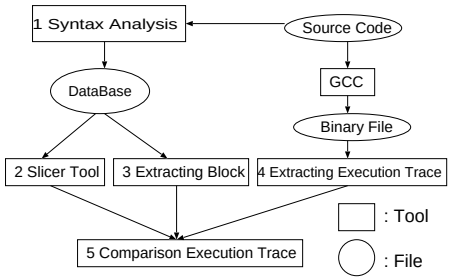

図 7 提案手法のフロー図
Fig. 7 Flow Graph of Proposed Method

号], [この行に依存している行番号へのポインタ配列], [この行が依存している行番号へのポインタ配列] である．このデータ構造により，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を抜き出し，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 を用いている．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が得られる．

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 節で述べたトレース比較を行う．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から，4.6 節で述べたバグ存在行の絞り込みを行う．

6. 性能評価

6.1 評価方法

評価に用いたプログラムを**表 2** に示す．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．それぞれのバグの内容を**表 3** に示す．

表 2 評価プログラム
Table 2 Programs for Evaluation

Pascal	入力された段数のパスカルの三角形を表示する
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
GA	入力された数の街を巡回する最適経路を遺伝的アルゴリズム (GA) で求める

表 3 作成されたバグ
Table 3 Generated Bugs

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

6.2 評価結果

表 3 に示したバグが存在するプログラムに対して，DPS と提案手法を用いた結果を**表 4** に示す．表中の DPS と提案手法は抽出された行数であり，削減率はソースコード全体から何%削減したかである．入力パターンはいずれの場合も，自然数 1〜10 である．

表 4 分岐命令の条件式バグ 評価結果
Table 4 Evaluation of Proposed Method

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 が平均約 79 %の削減率を示したのに対し，提案手法は平均約 94 %の削減率を示した．従って，DPS と比べて平均 15 %程度の削減率向上が得られた．

6.3 考 察

今回の評価では 7 例中 5 例において，バグの存在する行を特定できた．実際には，条件式の誤りではなく条件式で参照される変数値の算出にバグが存在しても同様の結果となり，その場合は抽出行にバグがないことになる．しかし，抽出された 1 行を調査する負担はわずかであり，その後，手作業による調査を行う場合も抽出行より前の実行部分を調べればよいため，作業量を削減することができる．

今回は特定の種類のバグのみ対象としているが，対象バグが存在しなくても 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．今後，他の種類のバグにも対応できれば，より効果を高められる．

この手法で取りこぼすバグとして，配列領域外アクセスなどによるデータ破壊が挙げられる．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．定数回ループ，ループ回数が入力値に比例など，各ループの特徴を静的解析によって検出できれば，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

本研究の一部は文部科学省科学研究費補助金（特定領域研究，研究課題番号 21013025 「タスクと実行環境の高精度モデルに基づくスケーラブルなタスクスケジューリング技術」）による．

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.htm.
2) Mireille Ducassé. A pragmatic survey of automated debugging. In *Proc. First International Workshop on Automated and Algorithmic Debugging*, pages 1–15, 1993.
3) R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32:57–95, 1987.
4) 網代 育大, 長 健太, and 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**, 15(1):54–58, 1998.
5) M.Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
6) H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25:246–256, 1990.
7) R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Software Eng. and Methodology*, 6:370–397, 1997.
8) 佐藤 慎一, 小林 孝則, 飯田 元, 井上 克郎, and 鳥居 宏次. 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. **電子情報通信学会技術研究報告**. *SS,* **ソフトウェアサイエンス**, 95(53):23–30, 1995.
9) 西松 顯, 楠本 真二, and 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. **電子情報通信学会技術研究報告**. *SS,* **ソフトウェアサイエンス**, 98(85):17–24, 1998.
10) Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 177–184. ACM, 1984.
11) Sapid. http://www.sapid.org/index-ja.html.
12) Xpath. http://docstore.mik.ua/orelly/xml/pxml/ch08_02.htm.