特別研究報告

題目

回帰テストにおける実行系列の差分の効率的な検出手法

指導教員
井上 克郎 教授

報告者
松田 直人

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

回帰テストにおける実行系列の差分の効率的な検出手法

松田 直人

内容梗概

ソフトウェア開発において，既存のプログラムに不具合，すなわちバグが見つかると，その修正作業が行われる．このとき，修正箇所は正常に動作するようになったものの，その修正がプログラムの他の部分にも影響を与え，新たなバグを生み出してしまうことがある．これを防ぐため，バグ修正後には，これまで成功していたテストも含めた回帰テストが行われる．

回帰テストでは，バグ修正前に成功していたテストケースがバグ修正後も成功することを確かめることで，新たなバグが発生していないことを確認する．しかし，実際にはバグが発生しているが，回帰テストは偶然通過してしまうということもありうる．そのため，バグ修正の際には，回帰テストによってプログラムの出力結果を検証するだけでは不十分であり，バグ修正前後でテストの動作そのものが変化していないことを確かめる必要がある．

プログラムの動作の差分を検出する技術として，バグ修正前後のテストの実行に対してそれぞれ動的依存グラフを構築し，そのグラフの差分を動作の差分として提示する手法が提案されている．しかし，グラフの比較方法に問題があり，実行される命令数が多くグラフが巨大になる場合に，正常に動作しないことがあった．

本研究では，動的依存グラフを有向辺の列に変換して比較することにより，スケーラビリティの改善を図った．また，実際の改善の程度を確認するために評価実験を行い，既存手法では検出できなかった差分についても検出できることを確認した．

主な用語

動的依存グラフ
フォワードスライス

目 次

1　まえがき

ソフトウェア開発において，既存のプログラムに不具合，すなわちバグが見つかると，その修正作業が行われる．このとき，修正箇所は正常に動作するようになったものの，その修正がプログラムの他の部分にも影響を与え，新たなバグを生み出してしまうことがある．これを防ぐため，バグ修正後には，これまで成功していたテストも含めた回帰テストが行われる．

回帰テストでは，バグ修正前に成功していたテストケースがバグ修正後も成功することを確かめることで，新たなバグが発生していないことを確認する．しかし，回帰テストはプログラムの出力結果のみを検証するため，プログラムの動作自体が変化していないことは確認できない．実際には動作が変化しているが，出力結果は偶然同じ値になるということもありうる．実際，Yin らの調査 [14] によれば，オペレーティングシステムの開発において，バグ修正の 14.8〜24.4%は別のバグを引き起こしており，修正版のリリース前には回帰テストが行われているにもかかわらず，バグを見落としていると報告されている．そのため，バグ修正の際には，回帰テストによってプログラムの出力結果を検証するだけでは不十分であり，テストの動作そのものが変化していないことを確かめる必要がある．

プログラムの動作を比較するための手法として，Ramanathan らは，プログラムの動作の違いをメモリ操作列の差分として検出する手法を提案した [8]．この手法では，プログラムの動作をメモリに対する読み出しと書き込みとして解釈し，プログラムの実行中に発生したメモリ操作の最長共通部分列を求めることで，動作の差分を提示する．

松村らは，メモリ操作以外の動作の違いも検出する手法として，プログラムの動的依存グラフを比較する手法を提案した [15]．この手法では，バグ修正前後のテストの実行に対してそれぞれ動的依存グラフを構築し，そのグラフの差分を動作の差分として提示する．しかし，動的依存グラフの比較方法に問題があり，実行される命令数が多くグラフが巨大になる場合に，正常に動作しないことがあった．

本研究では，松村らの手法における，動的依存グラフの比較アルゴリズムを変更することで，スケーラビリティの改善を図った．具体的には，グラフを有向辺の列に変換し，Myers のアルゴリズム [10] を用いて比較することで，動作の差分を検出する．また，実際の改善の程度を確認するために，評価実験を行った．評価実験では，松村らの手法と提案する手法を同じバグ修正前後のプログラムに対して適用し，動作の差分の検出結果を比較した．

以降，2 章では本研究の背景について説明し，3 章では提案手法を説明する．4 章で評価実験とその結果を示し，5 章で本研究のまとめを述べる．

2　背景

本章では，本研究の背景として，プログラムの実行系列の比較およびプログラム依存グラフについて述べる．また，先行研究として，松村らの研究について説明する．

2.1　プログラムの実行系列の比較

プログラムの実行を比較する手法については，これまで様々な目的で研究が行われてきた．Abramson らは，バージョンの異なる 2 つのプログラムの実行を比較する Relative Debugging を提案した [2]．Relative Debugging は，2 つのプログラムを，開発者が指定したデータ構造を比較しながら実行することで，差分を検出する技術である．この技術を用いると，旧バージョンでは正しく動作していたプログラムが，新しいバージョンで動作しなくなった場合などに，問題のある箇所を迅速に特定することができる．

Silva らは，プログラムの 2 つの実行系列のシーケンスアラインメントをとる手法を提案した [11]．この手法では，既存のシステムに新しい機能を追加するための手がかりとして，既に存在する機能を実行した際のメソッド呼び出しのアラインメントを提示する．これは，対象のシステムの実装を熟知していない開発者の労力の軽減に有用である．

Cornelissen らは，プログラムの実行系列をそれ自身と比較することで視覚化する手法を提案した [4]．この手法では，散布図の縦軸と横軸に同一の実行系列を割り当て，イベントごとに一定の類似基準を満たした座標にプロットすることで，実行系列の特徴を視覚化する．これにより，反復呼び出しや実行フェーズ，ポリモーフィズムの検出が可能となる．

本研究では，バグ修正前後のプログラムの実行を効率的に比較する手法を提案する．提案手法では，プログラムの動的依存グラフを有向辺の列に変換して比較することで，動作の差分を検出する．これによって，開発者は，バグ修正によって想定外の動作が起きていないことを確認することができる．

2.2 技術的背景

2.2.1 プログラム依存グラフ

プログラム中のある命令の実行が別の命令の実行に影響を与えるとき，その 2 つの命令間の関係は，依存関係と呼ばれる．依存関係には以下の 2 種類がある．

1. データ依存関係

命令 A，B が次の条件をともに満たすとき，A から B へのデータ依存関係が存在する．

- A がデータ d を定義し，B がそのデータ d を参照する．
- A から B への実行経路があり，その経路の少なくとも 1 つでは，データ d は他の命令によって再定義されない．

これは，A で定義されたデータ d が，他の命令によって上書きされることなく，B で参照される可能性があることを意味する．

2. 制御依存関係

命令 A，B が次の条件をともに満たすとき，A から B への制御依存関係が存在する．

- A は分岐命令である．
- A の結果によって，B が実行されるかどうかが決まる．

プログラム中の命令を頂点，データ依存関係・制御依存関係を辺とした有向グラフはプログラム依存グラフと呼ばれ [5]，ソースコードを静的に解析することで構築することができる．頂点が表す命令の粒度はグラフの利用目的によって様々であり，例えば丸山らの研究 [9] では，メソッド抽出リファクタリングを自動化するための解析として，基本ブロックを頂点としたプログラム依存グラフが利用されている．本研究では，プログラムの動作の詳細な差分を検出するために，最も細粒度な解析として，バイトコード命令を頂点とした動的依存グラフを構築する．例えば，図 1 の Java プログラムのバイトコード（図 2）に対してプログラム依存グラフを構築すると，図 3 のようになる．

```
1   public class Main {
2       public static void main(String[] args) {
3           int i, a = 1, b = 2;
4           if (args.length > 0) {
5               i = a + b;
6           } else {
7               i = a - b;
8           }
9           System.out.println(i);
10      }
11  }
```

図 1: サンプルコード

```
1   iconst_1
2   istore_2
3   iconst_2
4   istore_3
5   aload_0
6   arraylength
7   ifle            16
8   iload_2
9   iload_3
10  iadd
11  istore_1
12  goto            20
13  iload_2
14  iload_3
15  isub
16  istore_1
17  getstatic       #16         // Field java/lang/System.out:Ljava/io/PrintStream;
18  iload_1
19  invokevirtual   #22          // Method java/io/PrintStream.println:(I)V
20  return
```

図 2: 図 1 のプログラムのバイトコード

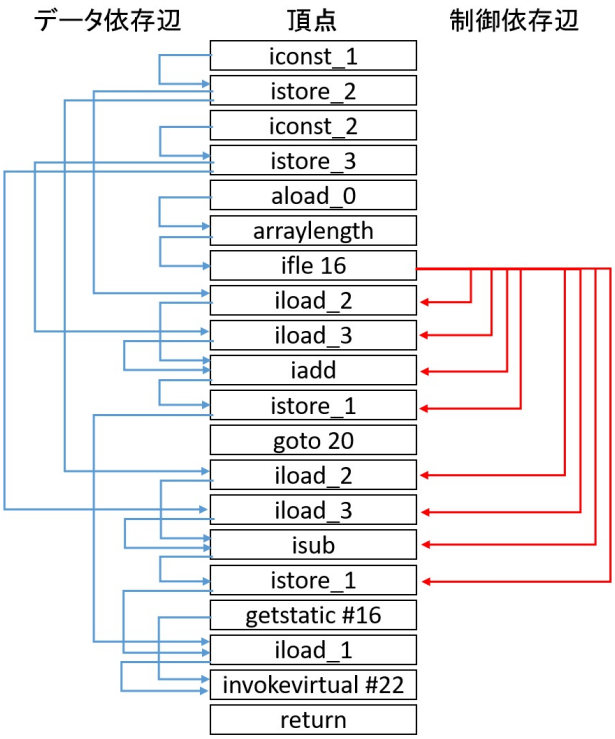

図 3: 図 1 のプログラムに対するプログラム依存グラフ

2.2.2 動的依存グラフ

プログラム依存グラフは，プログラムのいずれかの実行経路において成立しうる，すべての依存関係を表現している．それに対し，動的依存グラフは，ある特定の実行において成立した依存関係を表現する [3].

例えば，図 1 のプログラムがコマンドライン引数なしで実行された場合，3，4，7，9 行目の順に実行が進む．この実行に対して動的依存グラフを構築すると，図 3 のプログラム依存グラフのうち，図 4 に示した依存関係のみが抽出されることになる．なお，動的依存グラフにおいては，ループなどによって同一の命令が複数回実行される場合，それらは別々の頂点として扱われる．すなわち，実行された回数の分だけ，同一の命令を表す頂点が存在することになる．

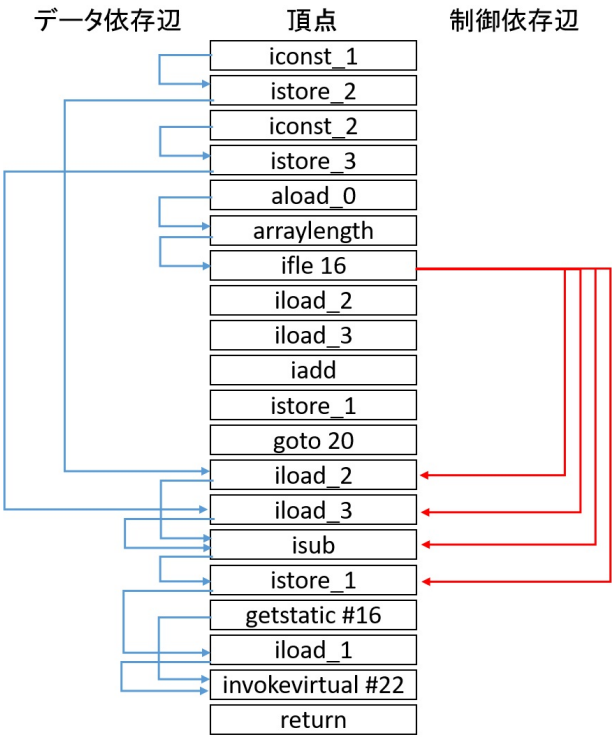

図 4: 図 1 のプログラムをコマンドライン引数なしで実行した場合の動的依存グラフ

2.2.3 プログラムスライス

プログラムスライス [13] は，プログラム中のある命令文に対して影響を与える可能性のある命令文の集合のことであり，プログラム依存グラフをもとに計算される．グラフ中から基準となる命令文（頂点）を選び，依存関係を逆方向にたどって到達可能な頂点集合を計算することで，その命令文に対して影響を与えうる命令文の集合を求めることができる．動的依存グラフに対して適用した場合は，プログラムの特定の実行において，その命令文に実際に影響を与えた命令文の集合が求められる．

一方，基準となる命令文の頂点から，依存関係を順方向にたどることで到達可能な頂点集合はフォワードスライスと呼ばれる．これは，プログラムスライスとは逆に，基準となる命令文のほうが影響を与える可能性のある命令文の集合を表す．

本研究では，バグ修正の前後で同一のテストケースを実行し，それぞれの動的依存グラフを構築する．その後，バグ修正が影響を与えた部分のみを抽出するために，修正したメソッド内の命令を基準としたフォワードスライスを計算する．バグ修正前後でフォワードスライスを比較することで，その修正によって発生した実行の差分を検出することができる．

2.3 先行研究

松村らの研究 [15] では，次の 4 つの手順にしたがって，バグ修正前後のプログラムの動作の差分を検出している．

1. プログラムの実行ログの記録
2. 動的依存グラフの構築
3. フォワードスライスの計算
4. フォワードスライスの比較

本研究では，このうち最後の手順である「フォワードスライスの比較」部分の手法を改善した．それ以前の「フォワードスライスの計算」までの手順については，松村らの手法をそのまま踏襲した．本節では，各手順の内容について簡単に説明する．具体例として，図 5 に示した Java プログラムを対象に動作の差分の検出を行う．

```
1  public class Main {
2      public static void main(String[] args) {
3          int i;
4          if (call(1, 2) > 0) {
5              i = 0;
6          } else {
7              i = 1;
8          }
9          System.out.println(i);
10     }
11     public static int call(int op1, int op2) {
12         return op1 + op2;
13     }
14 }
```

[a] 修正前

```
1  public class Main {
2      public static void main(String[] args) {
3          int i;
4          if (call(1, 2) > 0) {
5              i = 0;
6          } else {
7              i = 1;
8          }
9          System.out.println(i);
10     }
11     public static int call(int op1, int op2) {
12         return op1 - op2;
13     }
14 }
```

[b] 修正後

図 5: サンプルプログラム

2.3.1 プログラムの実行ログの記録

動的依存グラフを構築するための準備として，まず，Java プログラム用のロギングツールである SELogger[6] を用いて，プログラムの静的解析および動的解析を行う．

静的解析では，プログラム中のメソッド，バイトコード命令に番号をつける．メソッド番号はプログラム全体で一意な番号であり，バイトコード命令番号はそのバイトコード命令が存在するメソッド内で一意な番号である．よって，メソッド番号とバイトコード命令番号を組み合わせることで，プログラム中の任意のバイトコード命令を表現することができる．例として，図 5-a のプログラムのメソッド番号と，main メソッド内のバイトコード命令番号を表 1，2 に示す．表 1 中の`<init>`はコンストラクタを表す．

また，動的解析では，Java プログラムのバイトコードにログ記録用の命令を埋め込んで実行することで，メソッド呼び出し，フィールドアクセスなどのイベントや実行時の型情報を記録する．これにより，動的依存グラフを構築するための，プログラムの実行の再現が容易になる．

表 1: 図 5-a のプログラムのメソッド番号

番号	メソッド
0	`void Main.<init>`
1	`void Main.main(String[])`
2	`int Main.call(int, int)`

表 2: 図 5-a のプログラムの main メソッド内のバイトコード命令番号

番号	バイトコード
2	`iconst_1`
3	`iconst_2`
4	`invokestatic #16`
5	`ifle 13`
8	`iconst_0`
9	`istore_1`
12	`goto 15`
16	`iconst_1`
17	`istore_1`
21	`getstatic #20`
22	`iload_1`
23	`invokevirtual #26`
26	`return`

2.3.2 動的依存グラフの構築

記録した実行ログを補助情報として用いながら，対象のプログラムの実行を再現する．具体的には，クラスファイルからバイトコードを 1 命令ずつ読み込んで，その時点でのローカル変数，オペランドスタック，ヒープ領域上のオブジェクトの状態を復元する．このとき，各バイトコード命令に対してデータ依存関係および制御依存関係を計算することで，動的依存グラフを構築する．

1. データ依存関係の計算

各バイトコード命令 X に対し，次のようにしてデータ依存関係を求める．

- X が値 v を定義するとき，X を，v を定義した命令として記録する．
- X が値 v を参照するとき，v を定義した命令から X へのデータ依存関係が存在する．

例として，図 5-a のプログラムにおける動的依存関係の一部を図 6 に示す．main メソッド内の 9 番のバイトコード命令 `istore_1` は，int 型の値を 1 番目のローカル変数に格納する命令である．ここでは，ソースコードの 5 行目での変数 i の値を定義している．この値は，その後 22 番のバイトコード命令 `iload_1` によって参照される．したがって，9 番の命令 `istore_1` から 22 番の命令 `iload_1` へのデータ依存関係が存在する．

2. 制御依存関係の計算

また，各バイトコード命令 X に対し，次のようにして制御依存関係を求める．

- X が分岐命令であるとき，X をスタック S にプッシュする．
- スタック S のトップに分岐命令 Y が存在するとき，
  - Y の合流地点に到達したなら，Y をスタック S からポップする．
  - そうでないなら，Y から X への制御依存関係が存在する．

例えば，図 6 中の 5 番のバイトコード命令 `ifle` は，オペランドスタックの値が 0 以下の場合に指定された番地に制御を移す命令である．ここでは，ソースコード 4 行目の if 文を表している．この命令は分岐命令の 1 つであるから，実行時にスタック S にプッシュされる．その後，21 番のバイトコード命令 `getstatic` で合流するまでの間に，8，9 番の命令が実行されるので，5 番の命令 `ifle` から 8，9 番の命令 `iconst_0`，`istore_1` への制御依存関係が存在する．

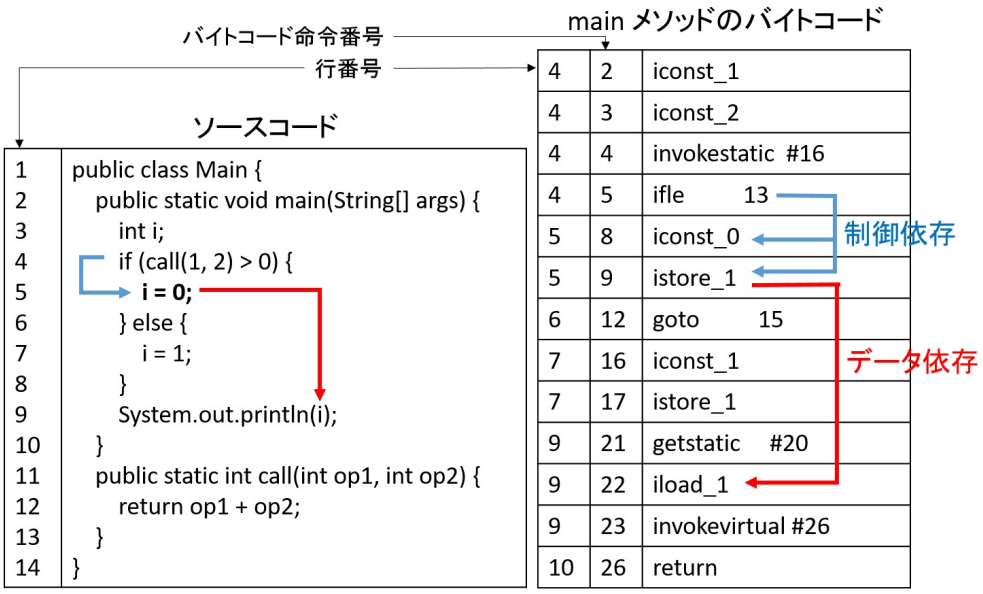

図 6: 図 5-a のプログラムにおける動的依存関係の例

以上のようにして計算した依存関係に基づいて，対象のプログラムの実行に対する動的依存グラフを構築する．グラフの頂点にはメソッド番号とバイトコード命令番号の組（表 1，2 参照）を使用することで，プログラム中の任意のバイトコード命令を表すことが可能である．

2.3.3 フォワードスライスの計算

構築した動的依存グラフをもとに，バグ修正を行ったメソッドを起点としたフォワードスライスを計算する．ただし，バグ修正を行ったメソッドは入力として与えられるものとし，バグ修正以外の変更は行っていないものとする．具体的には，グラフ中の依存辺を順に走査し，次の条件をともに満たしたとき，その依存辺および両端の頂点をフォワードスライスに加える．

- 依存辺の始点となる頂点が，バグ修正を行ったメソッド内の命令であるか，または，フォワードスライスに既に加えられている．
- 依存辺の終点となる頂点が，バグ修正を行ったメソッド内の命令ではない．

これによって，バグ修正を行ったメソッド内の命令からたどることができる依存関係のみが抽出される．すなわち，プログラムへの入力や現在時刻，乱数など，バグ修正以外の影響による動作の差分をできる限り排除することができる．

図 5-a のプログラムの，call メソッドを起点としたフォワードスライスを図 7（左）に示す．赤矢印がデータ依存辺，青矢印が制御依存辺を表す．図 7（右）は，わかりやすさのため，頂点のバイトコード命令番号を対応するバイトコードの Jimple 表現 [12] に置き換えたものである．

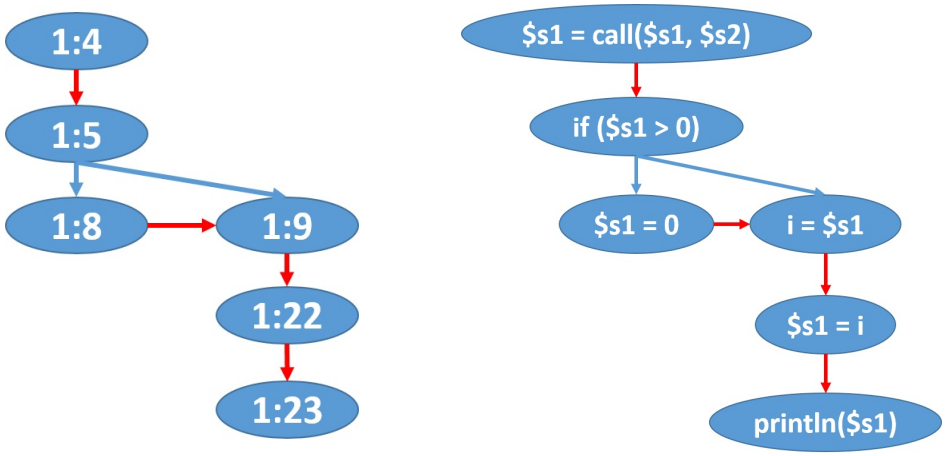

図 7: 図 5-a のプログラムの，call メソッドを起点としたフォワードスライス

2.3.4 フォワードスライスの比較

最後に，バグ修正前後のフォワードスライスを比較することで，プログラムの動作の差分を求める．フォワードスライスは有向非循環グラフに分類されるが，一般に，任意のグラフの差分を求める計算はコストが高く，巨大なフォワードスライスの比較には不向きである．松村らの手法では，その近似として，グラフ中の各頂点に対し，その頂点に到達しうるすべての経路上の頂点集合を計算することで，バグ修正前後での差分を検出している．

図 5 のプログラムに対するフォワードスライスを図 8 に示す．各頂点に付与された集合は，根からその頂点に到達しうるすべての経路上の頂点集合である．このとき，データ依存辺と制御依存辺の区別は行わない．例えば，修正前のフォワードスライスの 1:9 の頂点までの経路には，1:4，1:5，1:9 の経路と 1:4，1:5，1:8，1:9 の経路があるので，それらの経路上の頂点集合として {1:4，1:5，1:8} が付与される．これは，次のアルゴリズムで計算される．

アルゴリズム 1

1. 頂点のトポロジカルソートを行い，根から順に訪問する頂点リスト $V = [v_1, v_2, v_3, \cdots]$ を作成する．
2. 根から v_i に到達する経路上の頂点集合を $R(v_i)$ とし，$R(v_i) \leftarrow \phi$ で初期化する．
3. 各 v_i に対して，v_i から有向辺で接続されている頂点集合 $E(v_i)$ を列挙し，$v \in E(v_i)$ のそれぞれに対して，$R(v)$ を更新する．

$$R(v) \leftarrow R(v) \cup R(v_i) \cup \{v_i\}$$

頂点数を $|V|$，辺の数を $|E|$ とすれば，各頂点が最大で辺の数に等しい回数だけ $R(v)$ に加えられるので，このアルゴリズムの時間計算量は $\mathrm{O}(|V||E|)$ である．また，各頂点がそれぞれ頂点集合をもっているので，空間計算量は $\mathrm{O}(|V|^2)$ である．

このようにして付与された頂点集合を用いて，バグ修正前または修正後のフォワードスライスに固有な頂点を求める．ただし，修正前と修正後の頂点が同じメソッド番号，同じバイトコード命令番号をもち，かつその頂点までの経路上の頂点集合も同じ場合に限り，それらの頂点を同一の頂点とみなす．

例えば，図 8 のフォワードスライスでは，赤で表された頂点が差分として検出される．各差分の先頭の頂点 1:8，1:9 および 1:16，1:17 は，図 5 のプログラムにおける `i = 0;` と `i = 1;` に対応しており，バグ修正前後で異なる命令が実行されたことがわかる．

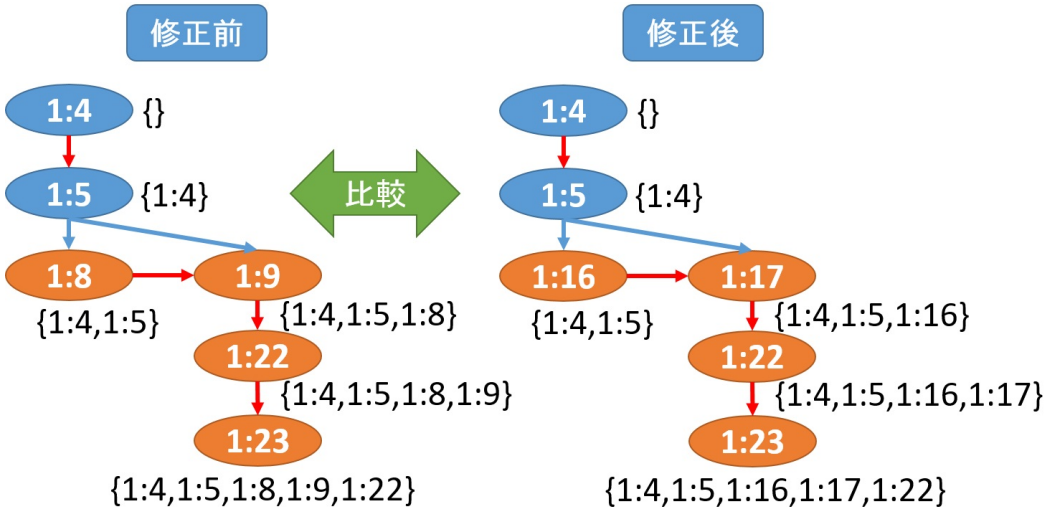

図 8: 図 5 のプログラムに対するフォワードスライスの比較

3　提案手法

本研究では，フォワードスライスの比較方法の改善として，Myers のアルゴリズム [10] を用いた手法を提案する．Myers のアルゴリズムは，2 つのシーケンスの差分を求めるアルゴリズムの 1 つで，文書比較プログラムの diff に利用されていることで知られている．提案手法は，次の手順にしたがって，バグ修正前後のフォワードスライスの差分を検出する．

1. 依存辺のファイルへの出力
2. 出力ファイルの比較

以降では，各手順の詳細について述べた後，松村らの手法との定性的な比較を行う．

3.1　依存辺のファイルへの出力

フォワードスライス内に含まれる依存辺を，図 9 に示すような形式でテキストファイルに出力する．1 行の文字列が 1 つの依存辺を表している．各行の出力形式は，以下のとおりである．

[依存辺の始点][依存関係の種類][依存辺の終点]

ただし，[依存辺の始点] および [依存辺の終点] は，その頂点が表す命令のメソッド番号とバイトコード命令番号を「:」でつないだ文字列であり，[依存関係の種類] は，データ依存のときは「D」，制御依存のときは「C」で表される．[依存辺の始点] がバグ修正を行ったメソッド内の命令の場合は，修正によってバイトコード命令番号がずれ，差分として誤検出されてしまうことを防ぐため，出力しない（図 9 の 1 行目）．

各依存辺の出力順は時系列順，すなわちその依存関係が成立した時刻の早い順とする．これは，データ依存関係の場合はデータを参照した時刻，制御依存関係の場合は分岐命令によって制御された命令が実行された時刻を意味する．なお，1 つの命令が実行されたときに，データ依存関係と制御依存関係が同時に成立した場合は，データ依存関係を先に出力するものとする．

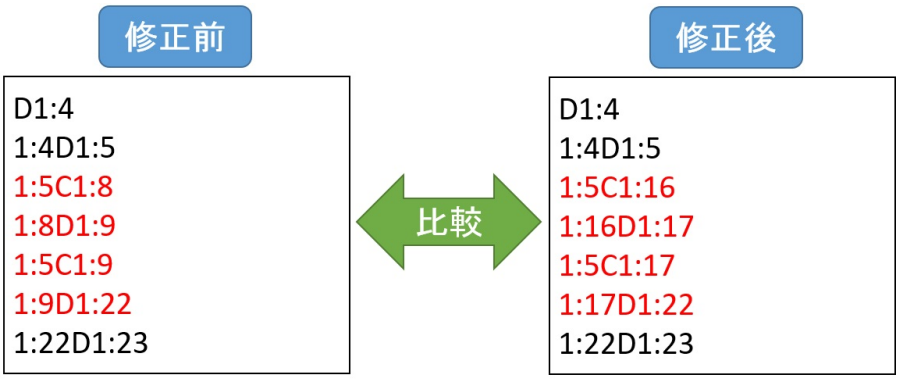

図 9: 図 5 のプログラムに対するフォワードスライスの出力

3.2 出力ファイルの比較

バグ修正前後のフォワードスライスの出力ファイルを Myers のアルゴリズムを用いて比較し，その差分を検出する．Myers は，2 つのシーケンスの差分，すなわち最長共通部分列（LCS）あるいは最短の変換手順（SES）を求める問題を，エディットグラフ上の最短経路問題に還元して解く方法を提案した [10]．2 つのシーケンスの要素数をそれぞれ $N1$，$N2$ とし，差分（編集距離）を D とすると，このアルゴリズムの時間計算量は $O((N1+N2)D)$，空間計算量は $O(N1+N2)$ である．

本研究では，Myers のアルゴリズムの実装として，GNU diff[1] を使用する．GNU diff は，2 つのテキストファイルを入力として受け取り，各入力ファイルを行単位で比較して，その差分を出力するプログラムである．本研究では，バグ修正前後のフォワードスライスの出力ファイルに対し，GNU diff を適用した．

図 9 の出力ファイルに対する適用結果を図 10 に示す.「<」で始まる行がバグ修正前に固有な依存辺,「>」で始まる行がバグ修正後に固有な依存辺である．バグ修正によって図 11 に示す 4 本の依存辺が変化していることがわかる．各差分の先頭の頂点 1:5 は，図 5 のプログラムにおける if 文に対応しており，バグ修正前後で異なる分岐が実行された結果，変数 i に異なる値が代入され，その後合流したことがわかる．

3.3 既存手法からの改善点

松村らの手法は，フォワードスライス中の頂点（命令）を，辺を考慮しながら比較し，その差分を出力する．一方，提案手法は，辺（依存関係）そのものを比較してその差分を出力する．この効果として，スケーラビリティの改善および検出精度の向上が挙げられる．具体

```
3,6c3,6
< 1:5C1:8
< 1:8D1:9
< 1:5C1:9
< 1:9D1:22
---
> 1:5C1:16
> 1:16D1:17
> 1:5C1:17
> 1:17D1:22
```

図 10: 図 5 のプログラムに対するに対するフォワードスライスの比較結果

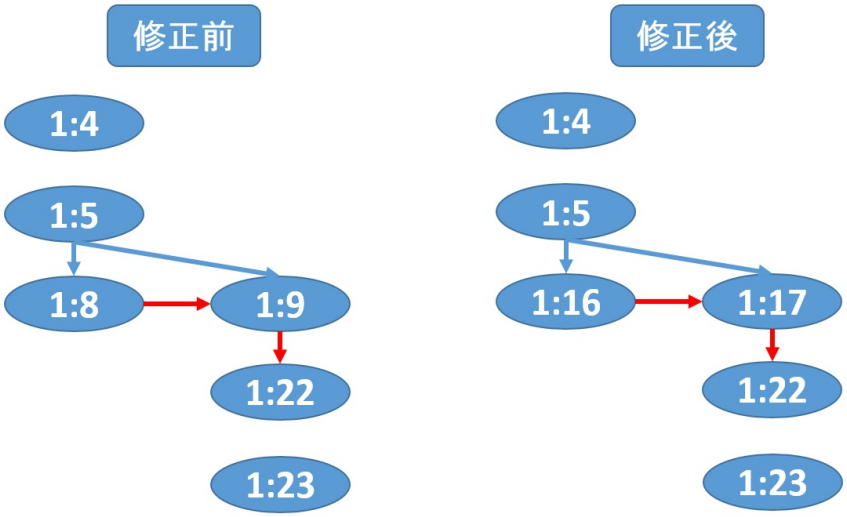

図 11: 図 5 のプログラムに対するフォワードスライスの差分

的な改善点を以下に述べる．

1. スケーラビリティの改善

バグ修正前後のフォワードスライスの頂点数を V_1，V_2，辺の数を E_1，E_2，それぞれに固有な辺の数を U_1，U_2 とすると，松村らの手法の時間計算量は $\mathrm{O}(V_1E_1 + V_2E_2)$，空間計算量は $\mathrm{O}(V_1^2 + V_2^2)$ である．一方，提案手法では，行数が $\mathrm{O}(E_1)$ と $\mathrm{O}(E_2)$ のファイルを入力とし，$\mathrm{O}(U_1 + U_2)$ の差分を出力するので，時間計算量は $\mathrm{O}((E_1 + E_2)(U_1 + U_2))$，空間計算量は $\mathrm{O}(E_1 + E_2)$ である．バグ修正前後のフォワードスライスは大部分が同じであると考えて，$V = V_1 \simeq V_2$，$E = E_1 \simeq E_2$，$U = U_1 \simeq U_2$ で近似すると，各計算量は表 3 のようになる．したがって，フォワードスライスが $\mathrm{O}(V) > \mathrm{O}(U)$ ないし $\mathrm{O}(V^2) > \mathrm{O}(E)$ であるならば，提案手法のほうが効率的に動作することが期待できる．これについて 4 章で評価実験を行い，実際の改善の程度を確認した．

2. 検出精度の向上

松村らの手法では，フォワードスライス中のある頂点に差分が見つかった場合，それ以降のすべての頂点が差分として検出される．そのため，プログラムの実行中に複数回の差分があった場合は，2 つ目以降の差分を特定できない可能性がある．提案手法では，diff を利用しているため，プログラムの実行に複数の差分がある場合でも，それぞれの差分を検出することが可能である．

また，松村らの手法では，バグ修正前後の頂点が同じメソッド番号，同じバイトコード命令番号をもち，かつその頂点までの経路上の頂点集合も同じ場合に，それらの頂点を同一の頂点とみなす．しかし，ループなどによって同一の命令が複数回実行された場合など，別の頂点であってもそれらすべてが一致してしまうこともある．提案手法では，データ依存辺と制御依存辺を時系列順に並べて比較するので，それらの場合でも正確に差分を検出することができる．

表 3: 各手法におけるフォワードスライスの比較の計算量

	時間計算量	空間計算量
松村らの手法	$\mathrm{O}(VE)$	$\mathrm{O}(V^2)$
提案手法	$\mathrm{O}(EU)$	$\mathrm{O}(E)$

4　評価実験

提案手法によってスケーラビリティが向上したことを確かめるため，評価実験を行った．実験には，松村らの研究と同様に，Defects4j[7] を利用した．Defects4j は，オープンソースソフトウェアのバグ修正に関するデータセットであり，バグ修正前・修正後のソースコードとそのテストケースによって構成されている．本実験では，Apache Commons Lang の 10 個のバグに対して，松村らの手法と提案手法をそれぞれ適用して，バグ修正前後の動作の差分検出を行い，その結果を比較した．実験は，Windows8.1（メモリ：256GB，CPU：Intel Xeon 2.90GHz）上で Ubuntu16.04（メモリ：241GB）の仮想環境を作成して行った．仮想化ソフトには VirtualBox5.0 を用いた．

フォワードスライスの計算までの処理時間を表 4 に，各手法によるフォワードスライスの比較結果を表 5 に示す．表 4，5 中の「(前)」「(後)」は，それぞれ「バグ修正前」「バグ修正後」を表す．また，表 5 中の「比較時間」には入出力のための時間は含まれない．すなわち，松村らの手法では「アルゴリズム 1 の実行時間」，提案手法では「diff プログラムの実行時間」のみを計測した．

実験の結果，松村らの手法では，4 番と 6 番のバグについて，メモリ不足により比較を完了することができなかった．一方，提案手法では，10 個すべてのバグに対して比較が正常に完了し，4 番を除く 9 個のバグにおける差分を検出することができた．4 番のバグについては，比較自体は正常に完了したものの，動的依存グラフに違いがないため，差分が検出されなかった．また，9 番のバグについては，動的依存グラフが比較的大きく，松村らの手法では比較に 500 秒以上の時間がかかっていた．しかし，提案手法では，4，6，9 番のバグのように，動的依存グラフが大きい場合においても短い時間で比較が完了している．

表 4: フォワードスライスの計算までの処理時間

番号	実行ログの記録 [s]	動的依存グラフの構築 [s]	スライスの計算 [s]
1 (前)	204	5611	1704
1 (後)	211	5774	1672
2 (前)	192	5049	1715
2 (後)	197	4924	1726
3 (前)	188	4813	1724
3 (後)	190	4916	1675
4 (前)	223	5841	2632
4 (後)	231	6400	2681
5 (前)	229	5461	1949
5 (後)	219	5583	1911
6 (前)	219	5292	4555
6 (後)	212	5254	2073
7 (前)	218	5539	1955
7 (後)	219	5408	1909
8 (前)	139	3627	875
8 (後)	136	3631	856
9 (前)	118	3078	743
9 (後)	115	3168	767
10 (前)	118	3152	631
10 (後)	115	3066	641

表 5: 各手法によるフォワードスライスの比較結果

	松村らの手法			提案手法		
番号	比較時間 [s]	頂点 [個]	固有頂点 [個]	比較時間 [s]	辺 [本]	固有辺 [本]
1 (前)	0.054	6260	0	0.002	11894	0
1 (後)		6996	42		13280	1386
2 (前)	0.016	1298	1	0.002	1410	2
2 (後)		1328	12		1470	62
3 (前)	0.059	6015	17	0.003	11430	62
3 (後)		6078	33		11552	184
4 (前)	N/A	51491332	N/A	3.423	104945779	0
4 (後)		51491332	N/A		104946779	0
5 (前)	0.008	330	1	0.001	429	1
5 (後)		396	19		508	80
6 (前)	N/A	83584442	N/A	43.744	165156523	0
6 (後)		83586850	N/A		165161272	4749
7 (前)	0.040	4769	5	0.002	9004	9
7 (後)		4868	5		9201	206
8 (前)	0.023	1580	0	0.001	2980	57
8 (後)		1583	33		2993	70
9 (前)	556.170	36795442	34	6.198	81033323	10729
9 (後)		36798484	19		81034003	11409
10(前)	1.250	631407	0	0.130	679846	2649
10(後)		631407	0		766242	89045

5　まとめ

本研究では，バグ修正によってプログラムに新たなバグが発生していないことの確認として，バグ修正前後における回帰テストの動的依存グラフを効率的に比較する手法を提案した．既存の手法では，実行される命令数が多く動的依存グラフが巨大になる場合に，差分を検出できないことがあった．提案手法では，グラフを有向辺の列に変換して比較することで，より大きなグラフに対しても差分の検出が可能となった．また，実際の改善の程度を調べるために評価実験を行い，既存手法では検出できなかった差分が検出できるようになったことを確認した．

今後の課題としては，出力結果の可読性の向上が挙げられる．diff プログラムの出力結果はバイトコード命令間の依存関係であるため，検出された差分がソースコード上のどの部分と対応しているかを開発者が確認することは困難である．したがって，出力結果をさらに変換することで，出力の可読性を高める必要がある．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には，研究に関する適切な御指導及び御助言を賜りました．井上教授の御指導及び御助言のおかげで本論文を完成させることができました．井上教授に心より感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には，研究において貴重な御意見を賜りました．多くの御指導及び御助言を頂いた松下准教授に深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教には，研究の各段階において多くの御助言を賜りました．多くの御指導及び御助言を頂いた石尾助教に深く感謝いたします．

最後に，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様には，本論文の執筆にあたって様々な場面で支えて頂きました．皆様のおかげで本論文を完成させることができました．心より感謝しております．

参考文献

[1] GNU Diffutils. `https://www.gnu.org/software/diffutils/`.

[2] David Abramson, Clement Chu, Donny Kurniawan, and Aaron Searle. Relative debugging in an integrated development environment. *Software Practice and Experience*, Vol. 39, No. 14, pp. 1157–1183, July 2009.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246–256, June 1990.

[4] Bas Cornelissen and Leon Moonen. Visualizing similarities in execution traces. In *Proceedings of the Workshop on Program Comprehension through Dynamic Analysis*, pp. 6–10, 2007.

[5] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the International Conference on Software Engineering*, pp. 392–411, May 1992.

[6] Takashi Ishio. SELogger. `https://github.com/takashi-ishio/selogger`.

[7] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 437–440, July 2014.

[8] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 241–252, September 2006.

[9] Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the International Symposium on Software Reusability*, pp. 31–40, November 2001.

[10] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, Vol. 1, No. 1, pp. 251–266, November 1986.

[11] Luciana Lourdes Silva, Klerisson Ribeiro Paixao, Sandra de Amo, and Marcelo de Almeida Maia. On the use of execution trace alignment for driving perfective

changes. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 221–230, March 2011.

[12] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 13–, 1999.

[13] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering*, pp. 439–449, March 1981.

[14] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 26–36, September 2011.

[15] 松村俊徳, 石尾隆, 井上克郎. 動的スライスを用いたバグ修正前後の実行系列の差分検出手法の提案. 情報処理学会研究報告, Vol. 2016-SE-191, No. 8, pp. 1–8, 2016/3/14.