修士学位論文

題目

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

指導教員
井上 克郎 教授

報告者
松村 俊徳

平成 28 年 2 月 9 日

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 ソフトウェア工学講座

平成 27 年度 修士学位論文

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

松村 俊徳

内容梗概

ソフトウェア保守において，現状のソフトウェアに対して変更を加えることが度々行われる．変更を加える目的は様々であるが，中でも特にバグ修正を目的とした変更は，是正保守と呼ばれている．是正保守において重視されるのが，修正対象であるバグに関連した動作以外にプログラムの動作に影響を与えないことである．そのため，プログラムに対して変更を加えた結果生じたプログラムの動作の変化を開発者が把握することが，新たに修正する必要のある箇所の特定や，変更による副作用が新たなバグを生んでいないことの確認に有用である．

プログラムの動作の変化は，単純にはカバレッジの上昇やテストの成功などにより開発者が見て取れる場合もあるが，一般にそうとは限らない．また，カバレッジの変化やテスト結果による判定では，動作の差分までを把握することは困難である．既存研究として，プログラムに加えた変更内容からその影響範囲を求める手法や，同一プログラムの 2 つの異なる出力からその原因となった入力を調査する手法などが提案されているが，これらは直接的にプログラム変更前後の動作の差分を求める手法ではない．

本研究では，修正前後のプログラムに同一の入力を与えて実行した場合に得られる 2 つの実行系列に対して，その動的スライスを比べることで，それらの差分を検出する手法を提案する．具体的にはプログラムの変更前後において，動的なプログラム依存グラフを構築しておき，変更を加えたメソッドを基点としたフォワードスライスを計算する．2 つのフォワードスライス内の各頂点に対して，その頂点に到達する経路上に存在する命令集合を比較することにより動作の差分を検出する．また，提案手法の有効性を調べるために，バグ修正に関する公開データセット Defects4j に含まれるアプリケーションに対して適用し差分を検出できることを確認した．

主な用語

動的解析
動的スライス

目 次

1　まえがき

ソフトウェア保守において，現状のソフトウェアに対して変更を加えることが度々行われる．その変更の目的にはバグの修正や，新たな機能の追加，パフォーマンスの改善，環境の変化に伴った修正などが挙げられる．ソフトウェア保守は，その動機や目的によって細分されており，中でも特に，発見されたバグに対する修正は，是正保守と呼ばれている．

是正保守において重視されるのが，修正対象であるバグに関連した動作以外にプログラムの動作に影響を与えないことである．そのため，バグ修正のためにプログラムに対して変更を加える際の開発者の関心事として，変更を加えることによりプログラムの動作にどのような変化がおきたかを知りたいというものがある．プログラムの動作の変化を把握することは，新たに修正する必要のある箇所の特定や，変更による副作用が新たなバグを生んでいないことの確認に役立てられる．

プログラムの動作の変化は，通常は回帰テストによって確認され，カバレッジの変化やテスト結果の変化として開発者が見て取れる場合もある．しかしながら，すでに他のテストケースによって網羅されている場合などでは，カバレッジに何の変化も起きず判断材料がないことになる．また，テストの成功により変化が見て取れた場合でも，対象となったテストケース以外で動作に変化がなかったことに対する保証はできない．

カバレッジの変化やテスト結果の変化以外にプログラムの動作の変化を見つける方法として Change Impact Analysis[3, 14, 9, 16] や Differencial Slicing[6]，Relative Debugging[1] などの手法が提案されている．Change Impact Analysis はプログラムに対して変更を加えることにより，プログラムの他のどの部分に影響を及ぼす可能性があるかを計算する手法である．しかしながら，あくまで影響が出る可能性があるかどうかを求める技術であるので，実際に変更を加えた後にどのような変化があったかまでを求めることはできない．Differencial Slicing は，プログラムの動作の差分を求める手法であり，プログラムに対して成功する入力と失敗する入力を与えることで，その差分がどこにあったかをプログラム依存関係レベルで検出できる．また，Relative Debugging は 2 つのプログラムを同時に実行しながらデバッグを行う手法で，双方のプログラムの変数値を同時に閲覧しながら確認を行うことができる．しかし，Differencial Slicing はプログラムの変更前後における動作の差分を求めるものではなく，入力の違いによる動作の差分を求める手法であり，Relative Debugging はあくまでデバッガであるため最終的に開発者の操作が必要となる．ゆえにこれらの手法をそのまま適用しただけでは，プログラム変更前後の動作の差分を求めることはできない．

本研究では，バグ修正前後における動的スライスを比べることで，プログラムの実行系列の差分を検出する手法を提案する．動的スライスは，プログラムの実行におけるデータの流れと実行制御を表現しており，それを比較することで実行系列の差分を検出する．具体的に

は変更前後のプログラムに対して同一の入力を与えて実行し，得られた実行系列から動的なプログラム依存グラフを構築しておき，変更を加えたメソッドを基点としたフォワードスライスを計算する．2つのフォワードスライス内の各頂点に対して，その頂点に到達する経路上に存在する命令集合を比較することにより動作の差分を検出する．

また，提案手法の有効性を調べるために，バグ修正に関する公開データセット Defects4j[7] に含まれるアプリケーションに対して適用した．10 個のバグに対して手法を適用したところ，6 個については差分が検出できることを確認した．

以降，2 章では，本研究の背景について述べる．3 章では，提案する手法について述べる．4 章では，実験について述べる．5 章では，関連研究について述べる．6 章では，本研究のまとめと今後の課題を述べる．

2　背景

本章では，背景として，バグ修正の副作用について述べる．また，提案手法は動的スライスを用いており，動的スライスは動的なプログラム依存グラフをもとに計算されるため，技術的な背景として，プログラム依存グラフとスライスについても述べる．

2.1　バグ修正の副作用

ソフトウェア開発において，テストの作成・実行により，潜在的なバグが見つかり，デバッグ作業が行われる．バグ修正が正しく完了したことは，テストの成功により確認される．しかしながら，ソフトウェアが複雑化・大規模化してくると，プログラムの相互関係を見落としてしまい，ある部分の修正が，一見何の関係もない別の部分に影響しバグを引き起こしてしまうことがある．このようなことが起きていないかの検証として，これまで成功していたテストを含めてテスト全体を再び実行する回帰テストが行われる．

回帰テストでは，変更前のプログラムでも実行が成功していたテストケースが，そのまま変更後も成功することを確認することにより新たなバグを生み出していないことの確認が行われる．しかしながら，テストの結果だけでは，動作が変化していないかどうかは判断できない．実際には動作が変化しているがテストは偶然成功しており，その動作の変化がバグに関連する可能性もある．実際に，Yin らの調査 [20] では，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%から 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リグレッション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．また，これらのバグの内 43%はシステムのクラッシュ，データ破壊，セキュリティ問題を引き起こすものであり，その影響は大きいと言える．

本研究では，動的スライスを用いてプログラムの動作の差分を検出する手法を提案する．テストの結果だけではなく動作の差分を提示することで，バグ修正により別のバグを生んでいるかもしれないという開発者に対する注意喚起が行え，また，差分が検出されなかった場合にはプログラム動作が同一であったことを保証できる．

2.2　プログラム依存グラフ

プログラム中の命令文の間には，データ依存関係と制御依存関係の 2 種類の依存関係が存在する．データ依存関係はプログラム中におけるデータの流れを制御依存関係は実行制御の流れをそれぞれ表現している．この 2 つの依存関係に関して，命令文を頂点，依存関係を辺として構成した有向グラフがプログラム依存グラフ (PDG) と呼ばれ [5]，プログラム中のデータ依存関係と制御依存関係を解析することで構築が可能である．

PDG の本来の定義はプログラム中の 1 命令を 1 頂点としているが，解析の目的に合わせて粒度を変更したグラフも使用されており，例えば，[11] ではメソッド抽出リファクタリングの自動化を目的として，基本ブロック単位の PDG が用いられている．本研究では，プログラムの動作の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単位での動的な PDG を構築する．

2.2.1 データ依存関係

データ依存関係は PDG を構成する辺の 1 つである．プログラム中の 2 つの文 s, t に対して，次の条件を満たすとき，s から t の間にデータ依存関係が存在する．

- s でデータ v を定義する
- t でデータ v を参照する
- s から t にデータ v を再定義しない実行経路が存在する

Java Virtual Machine(JVM) はスタックマシンとしてその動作が定義されており [17]，各命令が使用する一時的なデータを管理するためのスタック (以降オペランドスタックと呼ぶ) とローカル変数領域および Java ヒープ領域との間で，データのやりとりをしながら実行が進んでゆく．ゆえに，Java バイトコードにおけるデータ依存関係の抽出には，変数の定義，参照の他に，オペランドスタック上のデータに対する定義，参照も考慮する必要がある．

例として，図 1 に示す Java プログラムの Java バイトコード (図 2) に対して静的なデータ依存関係の抽出を行う．2 行目の *ICONST_1* 命令はオペランドスタックに対してデータをプッシュする命令であり，3 行目の *ISTORE* 命令はオペランドスタックのトップにあるデータをローカル変数領域に格納する命令である．つまり，2 行目の命令はオペランドスタック上のデータを定義し，3 行目の命令はオペランドスタック上のデータを参照していると言える．そして，この 2 つ命令の間でデータの再定義はないので，2 行目の *ICONST_1* 命令から 3 行目の *ISTORE* 命令に対してデータ依存関係が存在する．また，3 行目の *ISTORE* 命令はローカル変数の値を定義する命令でもあるので，ローカル変数を参照している 12 行目，19 行目の *ILOAD* 命令に対してデータ依存関係が存在する．このように，オペランドスタック，ローカル変数，ヒープ領域上のデータに対して，定義と参照の関係をもとにデータ依存関係は抽出される．図 3 に図 2 のバイトコード命令に対する静的なデータ依存関係をすべて抽出したものを示す．図 3 では，データ依存関係を青色の矢印で表現している．

```
 1 package dependency;
 2 public class Example {
 3         public static void main(String[] args){
 4
 5                 int i;
 6                 int a = 1;
 7                 int b = 2;
 8
 9                 if(args.length > 0){
10                         i = a + b;
11                 }else{
12                         i = a - b;
13                 }
14
15                 System.out.println(i);
16         }
17 }
```

図 1: サンプルコード

```
 1 (L00001)
 2 ICONST_1
 3 ISTORE 2 (a)
 4 (L00004)
 5 ICONST_2
 6 ISTORE 3 (b)
 7 (L00007)
 8 ALOAD 0 (args)
 9 ARRAYLENGTH
10 IFLE L00018
11 (L00011)
12 ILOAD 2 (a)
13 ILOAD 3 (b)
14 IADD
15 ISTORE 1 (i)
16 (L00016)
17 GOTO L00023
18 (L00018)
19 ILOAD 2 (a)
20 ILOAD 3 (b)
21 ISUB
22 ISTORE 1 (i)
23 (L00023)
24 GETSTATIC java/lang/System#out
25 ILOAD 1 (i)
26 INVOKEVIRTUAL java/io/PrintStream#println(I)V
27 (L00027)
28 RETURN
```

図 2: サンプルコードの Java バイトコード

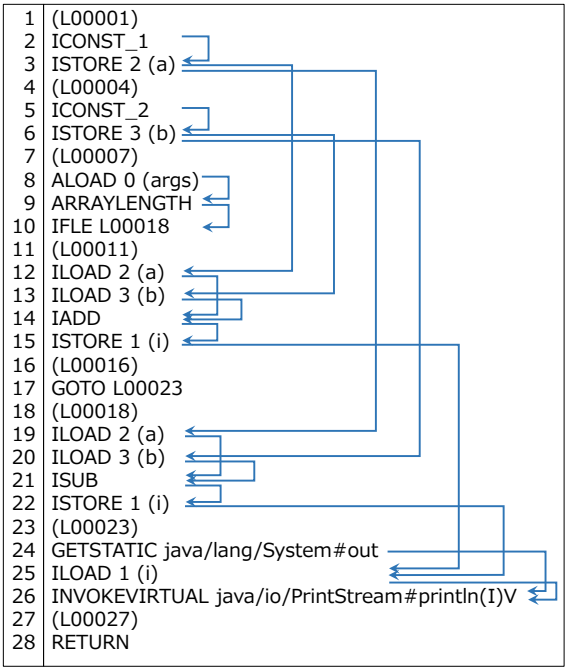

図 3: 静的データ依存関係

2.2.2 制御依存関係

制御依存関係は PDG を構成する辺の 1 つである．プログラム中の 2 つの文 s, t に対して，次の条件を満たすとき，s から t の間に制御依存関係が存在する．

- s は分岐命令である
- t の実行が s の判定結果に依存する

Java バイトコードにおける分岐命令は次の表 1 に示す 14 種類である．これらの命令から，*then* ブロック内, *else* ブロック内の各命令に対して，制御依存関係が存在する．図 4 に図 2 のバイトコード命令に対する静的制御依存関係を抽出した結果を示す．

2.3 動的なプログラム依存グラフ

動的なプログラム依存グラフ (動的な PDG)[2] は，プログラムの実行をもとに動的なデータ依存関係と動的な制御依存関係にもとづき構成したグラフである．静的な PDG では，ソースコードを解析し構築されるので，実際の動作ではありえないような依存関係も抽出される可能性がある．一方で，動的な PDG では，実際のプログラムの実行をもとに，依存関係を

表 1: 分岐命令一覧

オペランド 1 つ	オペランド 2 つ
IFEQ	IF_ICMPEQ
IFNE	IF_ICMPNE
IFLE	IF_ICMPLE
IFLT	IF_ICMPLT
IFGE	IF_ICMPGE
IFGT	IF_ICMPGT
IFNULL	
IFNONNULL	

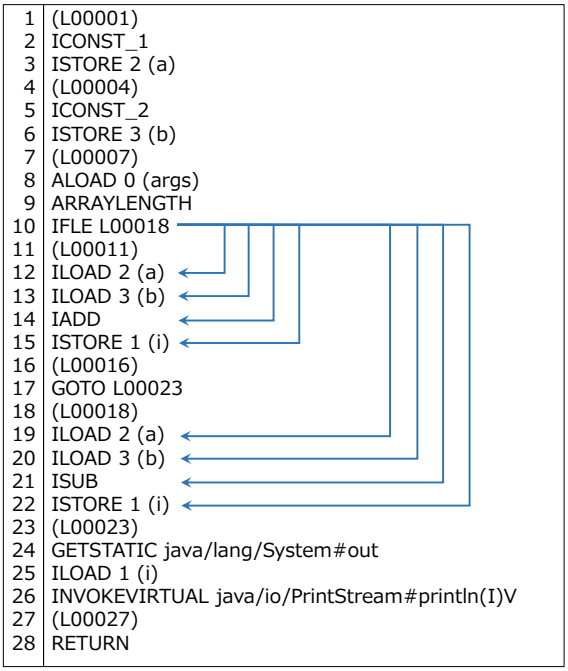

図 4: 静的制御依存関係

構築するため，実際の動作ではありえないような依存関係は存在しないという特徴がある．ただし，実行にもとづき実際に存在した依存関係を抽出するため，構築されるグラフは実行に依存したものとなる．

動的な PDG と静的な PDG では，グラフの頂点が表現しているものが異なる．静的な PDG の頂点は命令文であるのに対し，動的な PDG の頂点は時刻付きの命令文である．プログラムの実行においては，同一の命令を複数回実行することが考えられる．動的な PDG は実行にもとづき依存関係を抽出するため，同一命令が複数回実行されていた場合にはそれぞれを別頂点として扱いグラフが構築される．そこで同一命令を区別するために頂点に時刻情報が付与される．

静的なデータ依存関係は考えうる全ての実行経路に関して依存関係が抽出されるが，動的なデータ依存関係は実行をもとに実際に存在した実行経路に基づく依存関係のみ抽出される．図 1 に示したプログラムはコマンドライン引数の個数で実行経路が変化する．例えばこのプログラムに対してコマンドライン引数を空にして実行すると，図中の行番号で 5 → 6 → 7 → 9 → 12 → 15 と実行が進んでゆく．この実行をもとにデータ依存関係を抽出すると，図 5 に示す赤色の依存関係のみ抽出されることになる．また，動的データ依存関係と同様に，動的制御依存関係も実際に存在した実行経路に基づいて抽出され，図 5 と同じ実行をもとに動的制御依存関係を抽出すると図 6 のようになる．

2.4 プログラム依存グラフの利用法

プログラムスライスは Weiser らによって考案され [18]，プログラム中のある命令文に関して，その命令文に対して影響を与える可能性のあるすべての文のことであり，PDG をもとに計算される [5]．

プログラムスライスは，まず PDG 中からスライス基準となる命令文 (頂点) を選び，その頂点から逆方向にグラフを辿り到達可能な頂点集合を求めることで計算が可能である．静的な PDG をもとに計算すれば，基準とした命令文に影響を与える可能性のある命令文が，動的な PDG をもとに計算すれば，実際に影響を与えた命令文が抽出される．

プログラムスライスはデバッグ作業の支援などに利用されており，例えば，エラーを引き起こしている命令文を基準としてプログラムスライスを求めることにより，開発者はエラーに影響を与えた可能性のある部分にのみ焦点をあてることが可能となり，労力削減に貢献できる．他にも，計算の出力命令を基準とすれば，その計算がどのようにして行われるのか，計算に関係のある処理のみを抽出することができるため，プログラム理解の支援などにも用いられている．

プログラムスライスの計算では，ある命令文「に」影響を与える可能性がある命令文集合を，スライス基準となる命令文から逆方向にグラフを辿ることで計算していた．ここで，ス

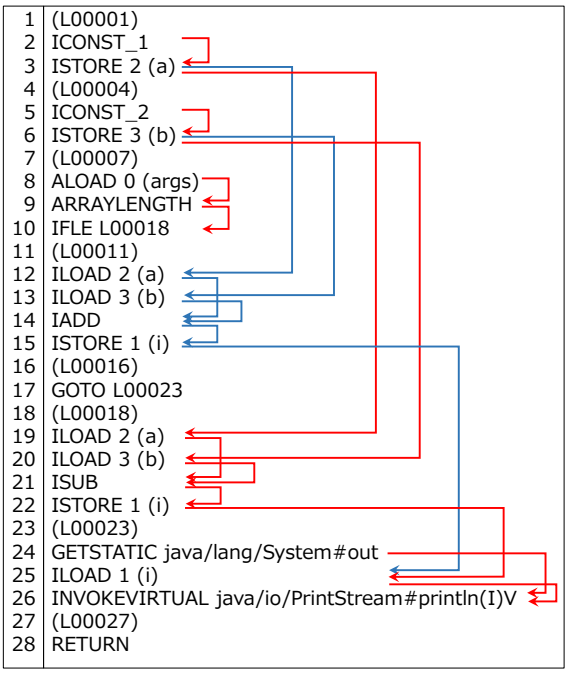

図 5: 命令間での動的データ依存関係

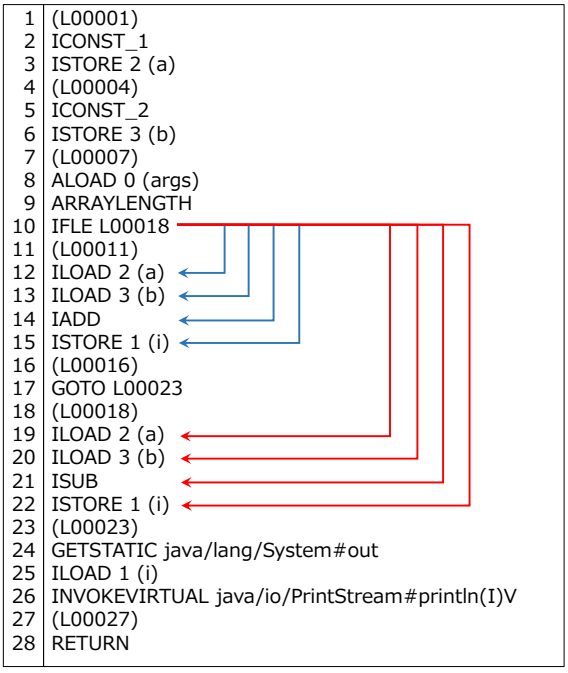

図 6: 命令間での動的制御依存関係

ライス基準となる命令文から，逆方向ではなく順方向にグラフを辿り，推移的に到達可能な頂点集合を求めることで，スライス基準となる命令文「が」影響を与える可能性がある命令文集合を求めることが可能である．このような，順方向にグラフを辿ってできるスライスをフォワードスライスと呼ぶ．さらに，動的な PDG をもとに変更地点からのフォワードスライスを計算すれば，スライス基準となる命令文が実際に影響を与えた命令文集合を求められる．

Li らはオブジェクト指向プログラムに対するフォワードスライスを用いた Change Impact Analysis 手法を提案しており [10]，継承やポリモーフィズムを考慮した上で，静的なデータ依存関係，制御依存関係をもとにフォワードスライスを計算し，影響を受ける可能性のある範囲を求めている．また，Orso らは動的情報を用いた Change Impact Analysis 手法を提案している [13]．この手法ではプログラムの実行を，プログラム中の各メソッドがプログラムの実行において呼び出されたか否かを表現したカバレッジ情報として記録している．カバレッジ情報をもとに，実行中に呼び出されたメソッドと，変更地点を基準として静的に計算したフォワードスライス内のメソッドとの共通メソッドを求めることで，擬似的にメソッドレベルの動的フォワードスライスを計算し，変更による影響範囲を検出している．

本研究では，変更前後のプログラムに対して同一のテストスイートを実行して動的な PDG を構築し，変更対象となったメソッドをスライス基準として，動的なフォワードスライスを計算する．そして，2 つのフォワードスライスを比較し動作の差分を検出する．

3　提案手法とその実装

本研究では，Java プログラムに対して，プログラム変更前後における動的スライスを比べることで，動作の変化を検出する手法を提案する．提案手法の概要としては，プログラムの変更前後において，動的プログラム依存関係を構築しておき，変更を加えたメソッドを基準としたフォワードスライスを計算する．２つのフォワードスライス内の各頂点に対して，その頂点に到達する経路上に存在する頂点集合を比較することにより動作の差分を検出する．提案手法は次に示す 4 つのステップから構成され，以降各ステップについての詳細を説明していく．

1. プログラムの実行ログの記録
2. 動的プログラム依存関係の計算
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

3.1　プログラムの実行ログの記録

Java プログラムの動的プログラム依存関係を計算するために，まず Java プログラムの実行の記録を行う．Java プログラムの実行ログの記録には Java プログラムのバイトコードに実行ログを書き出す命令を埋め込み，命令を埋め込んだプログラムを実行し実行ログを記録する．実行ログはイベントの列として表現し，各イベントには，それぞれのイベントの種類に応じて対応する値を記録している．イベントの種類の一覧を表 2 に示す．なお，表中のオブジェクト ID は，プログラム実行中に出現したオブジェクトに一意に割り当てた値である．

例えば，ReturnValue イベントというメソッドの返り値に関するイベントでは，返り値を記録し，PutField イベントというフィールドへの書き込みに関するイベントでは，書き込み対象のオブジェクト ID と書き込んだ値を記録する．例として，図 7 に示す Java プログラムの実行により記録されるイベントの列は表 3 のようになる．表中の ID1，ID2，ID3 はそれぞれ，String 配列型オジェクトの ID，PrintStream 型オブジェクトの ID，Test 型オジェクトの ID を表している．

実行ログの記録ではプログラム全体に命令を埋め込み，実行開始から終了までのプログラム全体の実行ログの記録を行っている．命令の埋め込みをプログラム全体ではなく，静的なプログラム依存関係を元に変更対象となるメソッドが影響を与えうる部分にのみ埋め込みを行うことで，プログラムの記録によるオーバーヘッドや，実行ログのサイズを削減することも可能である．しかしながら，一般に静的なプログラム依存関係から計算したフォワード

```
 1 package test;
 2 public class Test {
 3
 4       public static void main(String[] args) {
 5             Test obj = new Test();
 6             System.out.println(obj.square(10));
 7       }
 8
 9       public int square(int x){
10             int y = x*x;
11             return y;
12       }
13  }
```

図 7: サンプルコード

スライスは巨大であるため大きな効果は認められないことと，最終的に計算する動的なフォワードスライスのサイズは変わらないことから，本研究においては，プログラム全体に対して命令を埋め込み実行の記録を行う．

3.2 動的プログラム依存関係の計算

記録した実行ログをもとに，プログラムの実行の再現を行いながら動的プログラム依存関係の計算を行う．実行ログには次に示す情報が記録されていないため，バイトコードの命令を実行ログの情報を用いて実行しなおすことで，実行ログに記録されていない情報を復元する．

- バイトコード命令の実行順序
- 各命令を実行した時点でのローカル変数の状態
- 各命令を実行した時点でのオペランドスタックの状態
- 各命令を実行した時点でのヒープ領域上のオブジェクトの状態

そして，再現したローカル変数，ヒープ領域上のオブジェクトに対して，値の定義された時刻の情報を保持させておくことで，動的プログラム依存関係の計算を行う．

再現の方法としては，まず実行ログから最初のイベントを読み出す．そして，読み出したイベントに対応するバイトコード命令番号までの各バイトコード命令を適用し，ローカル変

表 2: イベントの種類

イベント名	イベントの意味	値
Entry	メソッドの開始	
FormalParam	仮引数	仮引数の値
NormalExit	メソッドの正常な終了	返り値
ExceptionalExit	メソッドの例外発生による終了	例外オブジェクト ID
Call	メソッドの呼び出し	
ActualParam	実引数	実引数の値
ReturnValue	メソッドの返り値の受け取り	呼び出し結果の返り値
GetField	フィールドの読み出し	読み出し対象オブジェクトの ID, 読み出した値
PutField	フィールドへの書き込み	書き込み対象オブジェクトの ID, 書き込んだ値
ObjectCreation	オブジェクトの生成	生成されたオブジェクトの ID
NewArray	配列の生成	生成された配列のオブジェクト ID, 配列の長さ
MultiNewArray	多次元配列の生成	生成された配列のオブジェクト ID, 配列の長さ
ArrayLoad	配列の値の読み出し	配列のオブジェクト ID, インデクス, 読みだした値
ArrayStore	配列の値の書き込み	配列のオブジェクト ID, インデクス, 書き込んだ値
ArrayLength	配列の長さの参照	配列のオブジェクト ID, 配列の長さ
Throw	throw 文による例外の送出	例外オブジェクト ID
Catch	catch ブロックによる例外の補足	捕捉したオブジェクト ID
InstanceOf	instanceof 命令の実行	対象のオブジェクト ID, 結果の boolean 値
MonitorEnter	synchronized ブロックの実行開始	同期対象のオブジェクト ID
MonitorExit	synchronized ブロックの実行終了	同期対象のオブジェクト ID

表 3: 図 7 のプログラムの実行により記録されるイベント列

イベント番号	イベント名	値	イベントの説明
0	Entry	ID1	main メソッドの開始
1	Call	-	Test クラスのコンストラクタの呼び出し
2	Entry	-	Test クラスのコンストラクタの開始
3	Exit	-	Test クラスのコンストラクタの終了
4	GetField	ID3	System クラスの out フィールドの読み出し
5	Call	ID2	square メソッドの呼び出し
6	ActualParam	10	メソッドの呼び出しの引数
7	Entry	ID2	square メソッドの開始
8	FormalParam	10	メソッドの呼び出しの引数
9	Exit	-	square メソッドの終了
10	ReturnValue	100	square メソッドの返り値
11	Call	ID3,100	println メソッドの呼び出し
12	ReturnValue	void	println メソッドの返り値
13	Exit	-	main メソッドの終了

数の状態，オペランドスタックの状態，ヒープ領域上のオブジェクトの状態を更新していく．バイトコード命令の適用時に，変数やオブジェクトの定義があればそれらの定義時刻を更新し，参照があればそれらが定義された時刻から現在時刻への依存辺を引く．対応するバイトコード命令番号まで各バイトコード命令を適用し終われば，次のイベントを読み出す．以上の処理をプログラムの終了イベントに達するまで繰り返す．なお，依存辺は始点と終点を表現した頂点の 2 つ組として記録し，頂点は時刻，使用されたメソッドに対し一意に割り当てたメソッド番号，バイトコード命令番号の 3 つの情報をもつ 3 つ組とする．時刻には，仮想的な時計として 1 つのバイトコード命令を処理する度にインクリメントされる値を用いる．

本研究では，一度実行を記録した後に記録を用いてプログラムの実行を再現し，動的プログラム依存関係の計算を行っている．しかし，動的プログラム依存関係を，プログラムの実行を行いながら計算する手法も存在する [12]. この手法を使うことで，プログラムの実行を記録してその後に再現する，という手順を踏まずにプログラム依存関係計算が可能である．しかしながら，小規模なプログラムであっても，動的なプログラム依存関係は多数出力されるため，計算した依存関係をメモリ上に保持し続けておくには限界があり，計算可能なプログラムのサイズに制限が存在する．そこで本研究では，プログラムのサイズに関する制限を小さくするために，プログラムの実行を記録し，実行を再現しながら依存関係の計算を行い，計算結果を書き出す方式を採用している．

3.2.1 データ依存関係の計算

データ依存関係は，ローカル領域，オペランドスタック，ヒープ領域，の 3 つの領域にある各データに対して，そのデータが定義された時刻を保持させておき，参照が起きた場合に定義時刻から現在時刻への辺を引くことで計算を行う．

表 4 にデータの定義，参照のあるバイトコード命令の一覧を示す．表中の “∗” 記号は各プリミティブ型を表すアルファベット，I，B，S，C，L，F，D と，リファレンス型を表す A を表現している．例えば，表中では，ILOAD, BLOAD, SLOAD, CLOAD, LLOAD, FLOAD, DLOAD, ALOAD をまとめて ∗LOAD と記述している．また，“+” 記号は等式・不等式を表すアルファベット EQ，NE，LE，LT，GE，GT，を表現しており，例えば，表中では，IFEQ, IFNE, IFLE, IFLT, IFGE, IFGT をまとめて IF+ と記述している．

また，表 4 中で使用している用語について，obj はリファレンス型のデータ，index は配列の読み書きの際に使用されるインデックス値，value は演算結果，params は引数，op はオペランド，local variable はローカル変数を，element は配列中の 1 要素を，それぞれ表している．

表 4: バイトコード命令一覧

バイトコード命令	オペランドスタックの変化	定義	参照
NEW	[...] → [..., obj]	obj	-
NEWARRAY	[..., index] → [..., obj]	obj	index
ANEWARRAY	[..., index] → [..., obj]	obj	index
MULTIANEWARRAY	[..., indexes] → [..., obj]	obj	indexes
ARRAYLENGTH	[..., obj] → [..., value]	value	obj
GETFIELD	[..., obj] → [..., value]	value	obj, field
GETSTATIC	[...] → [..., value]	value	field
PUTFIELD	[..., obj, value] → [...]	field	obj, value
PUTSTATIC	[..., value] → [...]	field	value
INVOKESTATIC	[..., params] → [..., value]	value	params
INVOKESPECIAL	[..., obj, params] → [..., value]	value	obj, params
INVOKEINTERFACE	[..., obj, params] → [..., value]	value	obj, params
INVOKEVIRTUAL	[..., obj, params] → [..., value]	value	obj, params
∗LOAD	[...] → [..., value]	value	local variable
∗ALOAD	[...] → [..., value]	value	local variable
∗STORE	[..., value] → [...]	local variable	value
∗ASTORE	[..., obj, index, value] → [...]	element	obj, index, value
INSTANCEOF	[..., op] → [..., value]	value	op
IINC	[...] → [...]	local variable	local variable
CONST∗	[...] → [..., value]	value	-
LDC	[...] → [..., value]	value	-
BIPUSH	[...] → [..., value]	value	-
SIPUSH	[...] → [..., value]	value	-
ATHROW	[...] → [..., value]	value	-
∗ADD	[..., op1, op2] → [..., value]	value	op1, op2
∗SUB	[..., op1, op2] → [..., value]	value	op1, op2
∗MUL	[..., op1, op2] → [..., value]	value	op1, op2
∗DIV	[..., op1, op2] → [..., value]	value	op1, op2
∗REM	[..., op1, op2] → [..., value]	value	op1, op2

次ページに続く

前ページの続き

バイトコード命令	オペランドスタックの変化	定義	参照
∗NEG	[..., op] → [..., value]	value	op
∗SHL	[..., op1, op2] → [..., value]	value	op1, op2
∗SHR	[..., op1, op2] → [..., value]	value	op1, op2
∗AND	[..., op1, op2] → [..., value]	value	op1, op2
∗OR	[..., op1, op2] → [..., value]	value	op1, op2
∗XOR	[..., op1, op2] → [..., value]	value	op1, op2
∗CMP	[..., op1, op2] → [..., value]	value	op1, op2
∗CMPG	[..., op1, op2] → [..., value]	value	op1, op2
∗CMPL	[..., op1, op2] → [..., value]	value	op1, op2
IF_ICMP+	[..., op1, op2] → [...]	-	op1, op2
IF+	[..., op] → [...]	-	op
IFNULL	[..., op] → [...]	-	op
IFNONNULL	[..., op] → [...]	-	op
∗2∗	[..., op] → [..., value]	value	op
∗RETURN	[..., op] → [..., value](caller stack)	value	op

3.2.2 制御依存関係の計算

既存研究において効率のよい動的制御依存関係の計算手法が提案されており [19]，その手法に倣って実装を行う．既存研究の手法では，スタック構造を利用して動的な制御依存関係を計算しており，以下にその基本的な方針を示す．

1. バイトコード命令の実行された順序に従って，バイトコード命令を 1 つずつ適用していく
2. 分岐命令に到達した場合，その命令をスタックにプッシュする
3. バイトコード命令の適用時にスタックのトップにある分岐命令がそのバイトコード命令を制御する
4. 分岐命令の合流地点にあたる命令文に到達したときにスタックから分岐命令をポップする

Java バイトコードにおける分岐命令とは表 1 に示した 14 種類であり，これら命令文がブロック内の命令文を制御することとなる．注意が必要なのは例外発生時の処理である．例外

```
 1 package dependency;
 2 public class ExceptionExample{
 3         public static void main(String[] args){
 4                 try{
 5                         call();
 6                 }catch(Exception e){
 7                         System.out.println("catch error");
 8                 }
 9         }
10
11         public static void call() throws Exception{
12                 int i = 1;
13                 if(i < 10){
14                         throw new Exception();
15                 }
16         }
17 }
```

図 8: サンプルコード：例外の発生

が発生した場合は，例外発生時にスタックのトップにある分岐命令が，例外がキャッチされた後の命令文を制御する．例えば図 8 のようなプログラムの実行を考える．なお，実際にはバイトコードレベルでの制御依存関係を構築するが，ここでは簡単のため Java ソースコードレベルでの制御依存関係の説明を行う．このプログラムは，5 行目→ 12 行目→ 13 行目→ 14 行目と実行が進んでゆき，14 行目で例外がスローされ，その例外がキャッチされて最後に 7 行目の命令が実行される．例外がスローされたときのスタックのトップは 13 行目の分岐命令であるので，例外がキャッチされた後の 7 行目の命令は 13 行目の分岐命令に制御依存することとなる．

3.3 フォワードスライスの計算

動的データ依存関係と動的制御依存関係をもとに，変更対象となったメソッドを基準としたフォワードスライスを計算する．一般にフォワードスライスは，動的データ依存関係と動的制御依存関係をもとに動的な PDG を作成し，基準となる頂点から推移的に到達可能な頂点集合を求めることで得られる．しかしながら，動的な PDG では，頂点数が実行時間に比例して増加していき，今回のような Java バイトコードレベルでの PDG では多くの頂点をもつこととなる．メモリ上に展開可能なグラフサイズにも限界があるので，本研究では動的データ依存関係と動的制御依存関係を時系列順に走査することで，動的な PDG 全体からフォワードスライスを切り出す形ではなく，基準点に依存辺を付け足していく形でフォワードスライスを計算する．具体的な計算方法としては，時系列順に依存辺を走査していき，次

のいずれかの条件を満たしたときに，グラフへの頂点と辺の追加を行う．全ての依存辺を走査し終えたときフォワードスライスの計算が完了する．

- 依存辺の始点となる頂点が，変更対象となったメソッド内の命令であり，かつ終点となる頂点が変更対象となったメソッド内の命令ではない
- 依存辺の始点となる頂点が，グラフ内に既に存在し，かつ終点となる頂点が変更対象となったメソッド内の命令ではない

3.4 フォワードスライスの比較

バグ修正前後の 2 つのフォワードスライスはいずれも有向非循環グラフであり，それらを比較することでプログラムの動作の差分を求める．しかしながら，フォワードスライスは実行の長さに比例して大きくなるため，単純なグラフの比較はコストが高く現実的ではない．そこで本研究では，フォワードスライス中の経路に着目して 2 つのフォワードスライスの比較を行う．

ただし，PDG は制御依存関係の存在により分岐と合流が多いのが特徴で，経路数を数え上げると組み合わせ爆発を起こしてしまう．そのため，グラフ中の経路集合の近似として，経路上の頂点集合の比較を行う．なお，フォワードスライス内における各頂点は，時刻，メソッド番号，バイトコード命令番号の 3 つの情報を保持しているが，時刻情報は 1 つのフォワードスライス内における相対的な情報であるため，2 つのフォワードスライスを比較する際には不要である．よって，フォワードスライスの比較には，時刻の情報を除外し，残る 2 つの情報だけを用いる．また，メソッド番号とバイトコード命令番号の組は，プログラム中の命令を一意に特定するものであり，以降 2 つの番号の組をプログラム命令番号と呼ぶ．

2 つのフォワードスライスの比較のために，まずフォワードスライス内の各頂点に対して，その頂点に到達する経路上の頂点集合を計算する．そして，2 つのフォワードスライスそれぞれに固有な頂点を差分として出力する．例えば，図 9,10 のような 2 つのフォワードスライスがあったとする．図中では，各頂点にプログラム命令番号を，頂点下部の波括弧にその頂点に到達する経路上の頂点集合を記述している．このとき，図中の青色の頂点は 2 つのフォワードスライス間で共通な頂点であり，赤色の頂点がそれぞれのフォワードスライスに固有な頂点である．

フォワードスライス内の各頂点に到達する経路上の頂点集合の計算は，次のように頂点を訪問しながら情報を伝播させる計算として定義する．

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

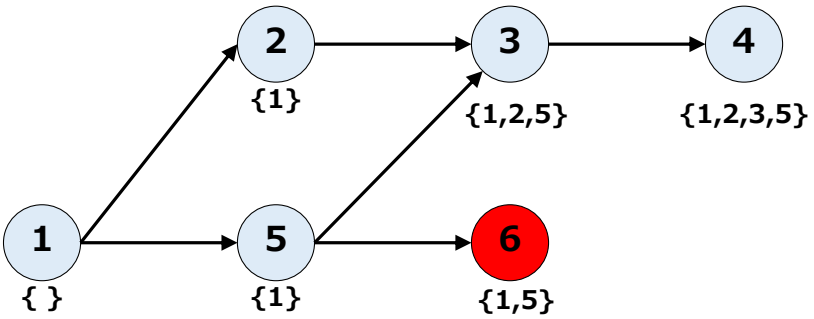

図 9: フォワードスライス 1

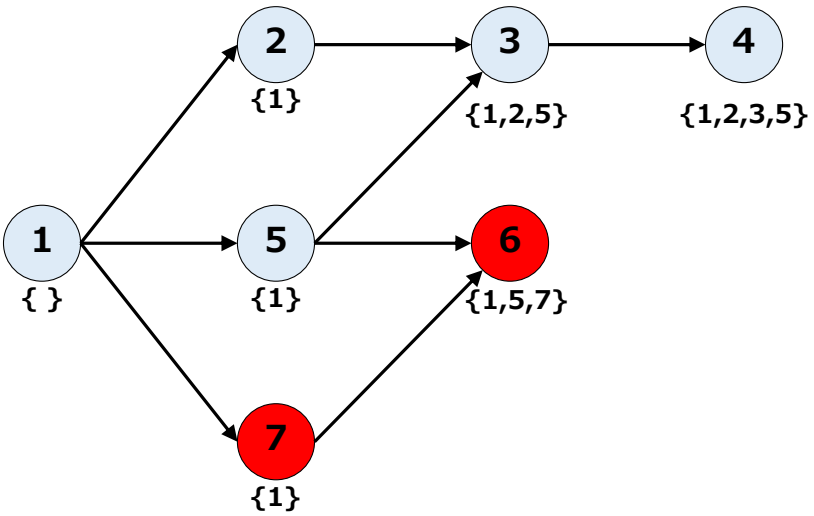

図 10: フォワードスライス 2

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，$v \in E(v_i)$ それぞれに対して，頂点 v_i のプログラム命令番号 $I(v_i)$ を用いて次のように $R(v)$ を更新する

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

この方法での頂点の訪問回数は $\mathcal{O}(|V|)$，情報の伝播回数は $\mathcal{O}(|E|)$ となり ($|V|, |E|$ はそれぞれ頂点数と辺の数)，最終的な計算時間は $\mathcal{O}(|V| \times |E|)$ となる．

2 つの頂点 $v1, v2$ が与えられたとき，頂点自身のプログラム命令番号と，頂点に到達する経路上の頂点集合がそれぞれ等しい場合に，2 つの頂点を等しいとみなす．これを式で表現すると以下のようになる．

$$v_1 = v_2 \Leftrightarrow (I(v_1) = I(v_2)) \wedge (R(v_1) = R(v_2))$$

2 つのフォワードスライスそれぞれに固有な頂点は，上記の頂点比較を用いた集合演算により計算する．2 つのフォワードスライス内の頂点集合 V_1, V_2 に対して，それぞれに固有な頂点集合 U_1, U_2 を次の式で求める．

$$U_1 = V_1 - V_2$$

$$U_2 = V_2 - V_1$$

4　実験

提案手法の有効性を評価するために実験を行った．実験にはバグ修正に関する公開データセット Defects4j[7] 内のアプリケーションを使用した．実行環境は，OS が Ubuntu14.04，メモリが 256GB，CPU が Intel Xeon 2.90GHz である．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，修正前・修正後のソースコードと，エラーを引き起こすテストケースが含まれている．実験では，その中でも Apache Commons Lang と呼ばれるオープンソースソフトウェアの 10 個のバグに対して，全テストケースの実行を記録し，バグ修正前後の動作の差分検出を行った．

表 5 に各手順における計算時間を，表 6 に抽出したフォワードスライスの規模および差分として検出された頂点数を示す．表中の記号 b はバグ修正前，f がバグ修正後を表してる．例えば 1b であればバグ番号 1 番のバグ修正前のソースコードを示している．表 6 において，経路はフォワードスライス中の入次数が 0 の頂点から，出次数が 0 の頂点に到達する経路の総数である．また，固有な頂点は，バグ修正前後のフォワードスライスの差分として検出された頂点数を示している．

表 6 より，8 個のバグについて，その修正前後の差分を検出でき，残る 2 つのバグはメモリ不足により，計算が完了しなかった．1 番のバグでは，フォワードスライスの頂点数がおよそ 6 千，辺の数がおよそ 1 万，経路数は約 357 京とあるように巨大なグラフとなっている．このような巨大なグラフに対しても，本手法を適用することで差分を検出することができ，検出された差分としては，バグ修正後に新たに 42 の頂点が加わっていることが確認できる．

しかしながら，バグ番号 8 のバグはフォワードスライスのサイズが 0 という結果になっている．これは，実行の記録対象外における依存関係は抽出できないためである．実行の記録に含まれないクラスや，そのオブジェクトに対するメソッド呼び出しについては，メソッド内部の処理が観測できない．ゆえに，メソッド内でフィールドへの書き込みによりデータの定義があり，メソッド外でそのフィールド値を参照していた場合などに，その依存関係が抽出されず，結果的にフォワードスライスを計算した場合に依存辺が途切れる形となる．本研究においては，Java クラスライブラリ内のクラスに関しては実行の記録をしていないため，これらのクラスのメソッド内外における依存関係が抽出されず，このような結果となっている．

このような状況を避けるための対策としては 2 つ考えられる．1 つは，Java クラスライブラリ内のクラスに関しても実行を記録することである．そうすることで，すべての依存関係を正しく抽出することができる．もう 1 つは，Java クラスライブラリ内のオブジェクトに対するメソッド呼び出しがあった場合には，呼び出しによってオブジェクト内のフィールドが定義される可能性があるとして，呼び出し命令自体をオブジェクトを定義する命令として

扱うことである．これにより，メソッド内部の処理を観測できない場合でも，依存辺が途切れることがない．ただし，実際には依存関係がない場合にも，依存関係があるものとして抽出されてしまい，正しくデータの流れや制御の流れを追えなくなるという欠点がある．

表 5: 各手順の計算時間

バグ番号	実行ログの記録 [s]	動的 PDG の計算 [s]	スライス計算 [s]	スライス比較 [s]
1b	185	5,507	884	0.019
1f	187	5,261	760	0.015
2b	184	5,429	763	0.004
2f	185	5,374	762	0.001
3b	186	5,293	796	0.014
3f	184	5,174	662	0.010
4b	187	5,482	1,189	N/A
4f	191	5,372	1,637	N/A
5b	186	5,274	1,234	0.003
5f	185	5,286	1,224	0.001
6b	189	5,320	2,546	N/A
6f	188	5,171	2,916	N/A
7b	189	5,470	1,231	0.022
7f	185	5,439	1,203	0.012
8b	91	2,601	366	0.001
8f	93	2,565	350	0.001
9b	104	2,932	1,272	1,231.148
9f	104	2,895	1,918	1,909.242
10b	104	2,906	427	0.616
10f	107	2,795	416	0.400

表 6: フォワードスライスの情報

バグ番号	頂点 [個]	辺 [本]	経路 [通り]	固有な頂点 [個]
1b	6,260	11,024	357×10^{16}	0
1f	6,996	12,321	357×10^{16}	42
2b	1,298	1,364	319	1
2f	1,326	1,404	337	7
3b	6,015	10,591	357×10^{16}	0
3f	6,078	10,705	357×10^{16}	3
4b	51,491,332	97,249,586	913×10^{16}	N/A
4f	51,491,332	97,249,586	913×10^{16}	N/A
5b	329	389	136	0
5f	396	462	167	19
6b	83,584,442	156,272,662	201×10^{16}	N/A
6f	83,586,850	156,277,095	206×10^{16}	N/A
7b	4,603	8,002	357×10^{16}	39
7f	4,812	8,373	357×10^{16}	73
8b	0	0	0	0
8f	0	0	0	0
9b	68,481,669	119,895,430	532×10^{16}	2
9f	68,481,483	119,895,112	532×10^{16}	0
10b	631,407	631,407	131,313	0
10f	631,407	631,407	131,313	0

5　関連研究

Johnson らは 2 つの動的プログラムスライスをもとに，その違いを検出する手法を提案している [6]．この手法では，テストの成功パターンと失敗パターンを与え，それぞれに対して動的プログラムスライスを求めて比較し，成功と失敗の違いを生んだ命令文を提示している．プログラムスライスの比較においては，ポインタの正規化などにより枝刈りを行い探索空間の削減することで実用性を高めている．我々の手法は，プログラムの変更による動作の違いを検出する目的であり，入力となるテストスイートは同一であるが，対象となるプログラムが変化している．ゆえに，バックワードのスライスではなくフォワードスライスを用いている点がこの手法とは異なる．

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 [1]．Relative Debugging は 2 つのプログラムに同じ入力を与えて実行を比べることで欠陥を特定する技術であり，開発者が 2 つのプログラム間で状態が等しくなるべき地点を指定すると，状態が異なっていた場合にそのことが通知される．この技術はソフトウェア開発などで，正しく動作していた旧バージョンのプログラムに改変を加えたところ，新バージョンでは正しく動作しなくなった場合などに，旧バージョンと新バージョンの実行を比べることで，効率的なデバッグが可能となる．Relative Debugging はあくまでインタラクティブなデバッガであり，動作の差分を検出するのに開発者の操作を必要とするが，我々の手法は変更されたメソッドを入力として，自動的に差分を検出する手法である．

Ramanathan らは動作の違いをログの最長共通部分列を求めることで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，読み出し・書き込み操作の列を求める．プログラムの変更前後の 2 つの列の最長共通部分列を求め，動作の差分を提示する．この手法はメモリに対する操作にのみ着目するのに対し，我々の手法は，メモリに対する読み出し・書き込み以外の命令に関しても依存関係に差分があれば検出する点が異なる．

Cai らは dynamic impact analysis の正確さを推定するための手法を提案した [4]．この手法では，プログラムの実行中に実行された命令文とその時の値の列を比べることでメソッドレベルでの動作の変化を検出している．我々の手法では，より細粒度な Java バイトコードレベルでの動作の変化を検出している．

Perscheid らはステートナビゲーションを提示することでシステマティックに不具合から欠陥へとたどり着けるような手法を提案した [15]．この手法は，テストケースの実行をもとにオブジェクトの正常状態なるものを構築しておき，失敗するテストに対して，正常状態かどうかのチェックを行い，怪しい部分を開発者に提示している．すなわち，オブジェクトの状態という観点で実行を比較して差分を検出していると言える．我々の手法は動的なプログ

ラムスライスのみを用いて動作の差分を検出しているが，この手法を応用し，オブジェクトの状態という情報とプログラムスライスを組み合わせることで，動的スライスの比較の際に枝刈りを行え，効率的にプログラムの動作の違いを検出可能かもしれない．

6　まとめと今後の課題

本研究では，バグ修正前後のプログラムの実行系列の差分を検出する手法を提案し，実装を行った．

手法は修正前後において，動的なプログラム依存グラフを構築しておき，変更を加えたメソッドを基点としたフォワードスライスを計算し，2 つのフォワードスライスを比較するものである．フォワードスライスはグラフとなるため，その比較コストは高い．提案手法では２つのフォワードスライス内の各頂点に対して，その頂点に到達する経路上に存在する頂点集合を比較することで，これを解決している．また，提案手法の有効性を調べるために，Defects4j に含まれるアプリケーションの 10 個のバグに対して適用し，6 個のバグに関して，その修正前後の動作の差分を検出できることを確認した．

今後の課題としては，計算の高速化とスケーラビリティの向上が挙げられる．本研究では，プログラム全体の実行を記録した後に，再びプログラムの実行を再現しながら，動的な PDG の計算を行っている．プログラム全体の実行を記録するのではなく，変更対象となったメソッドを基準とした静的なフォワードスライスの範囲のみプログラムを記録することで，実行の記録，動的な PDG の計算，フォワードスライスの計算にかかる時間を削減できる可能性がある．また，プログラム修正前後の 2 つの動的なフォワードスライスの大部分は差分がないはずである．この情報を活用し，差分がない部分のグラフ構造を 2 つのフォワードスライスで共有し，必要なメモリ量を削減することでスケーラビリティの向上が考えられる．

謝辞

本研究において，常に適切な御指導および御助言を賜りました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心より深く感謝いたします．これまで，研究生活において当たり前のように計算機器を利用することがすることができました．振り返ってみれば，それはとても恵まれた環境だったように思います．学部生の時からこれまでの計 3 年間，非常に快適な研究生活を送ることができたこと，重ね重ね感謝いたします．

本研究において, 適切な御指導をして頂き，また，計算機の管理面でも逐次適切な御助言を頂けた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深く感謝いたします．

本研究において, 研究の背景から実装に至るまで，非常に細部においても随時適切な御指導および御助言を頂きました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深く感謝いたします．研究に行き詰った際や，研究発表の際などに頂けたその御指導は私にとって非常に助けとなりました．

本研究において, 様々な御指導および御助言を頂きました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神田哲也氏に深く感謝いたします．些細なことまで相談に乗り丁寧な対応を頂けたこと，深く感謝いたします．

最後に，その他様々な御指導，御助言等を頂き，研究生活の支えとなっていてくれた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深く感謝いたします．

参考文献

[1] 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.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN90 Conference on Programming Language Design and Implementation*, pp. 246–256, June 1990.

[3] Haipeng Cai and Raul Santelices. A framework for cost-effective dependence-based dynamic impact analysis. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, pp. 231–240, March 2015.

[4] Haipeng Cai, Raul Santelices, and Tianyu Xu. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis. In *Proceedings of the International Conference on Software Security and Reliability*, pp. 48–57, June 2014.

[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] Noah M. Johnson, Juan Caballero, Kevin Z. Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the International Conference on Security and Privacy*, pp. 347–362, May 2011.

[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] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pp. 308–318, May 2003.

[10] Li Li and A.Jefferson Offutt. Algorithmic analysis of the impact of changes to object-oriented software. In *Proceedings of the International Conference on Software Maintenance*, pp. 171–184, November 1996.

[11] 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.

[12] Vijay Nagarajan, Dennis Jeffrey, Rajiv Gupta, and Neelam Gupta. A system for debugging via online tracing and dynamic slicing. *Software Practice and Experience*, Vol. 42, No. 8, pp. 995–1014, July 2012.

[13] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 128–137, September 2003.

[14] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the International Conference on Software Engineering*, pp. 491–500, May 2004.

[15] Michael Perscheid, Tim Felgentreff, and Robert Hirschfeld. Follow the path: Debugging state anomalies along execution histories. In *Proceedings of the International Conference on Software Maintenance, Reengineering and Reverse Engineering*, pp. 124–133, February 2014.

[16] Xiaoxia Ren, Barbara.G Ryder, Maxmilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the International Conference on Software Engineering*, pp. 664–665, May 2005.

[17] Lindholm Tim, Yellin Frank, Bracha Gilad, and Buckley and Alex. Java virtual machine specification. `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

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

[19] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 185–195, July 2007.

[20] 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.