Vol.2022-SE-210 No.25
2022/3/12

実行トレースのマークル木を用いたプログラム変更前後の差分検出法の提案

成　泰鏞[1,a)]　石尾 隆[1]　松本 健一[1]

概要：ソフトウェア保守において, ソフトウェアはデバッグ, 機能追加などの理由により変更が加えられる．開発者は, プログラム変更前後の動作の変化を把握することが重要である．Omniscient Debugging に用いる実行トレースは, プログラムの実行開始から終了までの命令が網羅的に記録されている．実行トレースから詳細な実行系列を把握することができる．一方で, 実行トレースは膨大なデータ量となることがあり, ソフトウェア変更前後の実行トレースを直接比較することは困難である．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実行トレースに対して, ハッシュ値を用いて実行トレースを要約し, そのハッシュ値を用いた差分検出手法を提案する．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．

1. はじめに

ソフトウェア保守は開発者にとって難しい作業の 1 つである．開発者は，バグ修正や機能の追加，ソフトウェアのパフォーマンス改善などの様々な理由からプログラムに変更を加えるが，そのようなプログラムの変更が変更箇所とは異なる部分に影響することで，開発者が予期しない動作を引き起こすことがある．プログラムに変更を加えるとき，変更による影響や，変更後のプログラムの動作を把握することが重要である．

変更後のプログラムの動作を確認するために，開発者はソフトウェアを実行する [1]．変更前後のプログラムに対して，同一の入力データを与えて実行し，その結果を比較することで，変更による影響を確認することができる．しかしながら，テストの結果だけでは，対象となったテストケースで注目する出力値以外の動作に変化がなかったことを保証することはできず，変更による影響を把握することは困難である．同様に，カバレッジなどのテストの網羅率の変化によってプログラム変更前後の影響が見て取れる場合があるが，プログラムの変更次第で網羅率が大きく変化することも考えられるため，正確に影響を把握することは難しい．

ソフトウェアの動作を収集する手法としてロギングがある．通常のロギングでは，様々な処理の進行状況や，例外的な動作の発生を，メッセージの系列として外部に出力するよう，プログラムにあらかじめ命令を埋め込んでおく [2]．これらに加えて，外部からプログラムの動作を観測するトレーシングツールも存在しており，たとえば Omniscient Debugging[3] は，ある入力におけるプログラムの全実行系列を実行トレースとして保存することで，プログラムの任意の時点の変数の状態を分析する手法である．実行トレースには，プログラム実行の詳細な情報が記録されるが，膨大なデータ量が記録されることがある．

実行トレースを比較することでプログラムの動作の差異を分析することが可能である．たとえば Relative Debugging [4] は，異なる版のプログラム 2 つを並行に実行し，その動作を照合して，異なる動作（実行される命令やメモリの値）が観測された時点でプログラムを一時停止し，デバッガに制御を移して分析を可能とする手法である．しかし，本研究のようにバグ修正の結果を確認しようとする場合は，修正対象コードの実行位置で必ず差異が発生して実行が一時停止すると期待されるため，それ以降の動作の違いは手作業で分析する必要がある．実行結果の違いを比較する手法としては，同一バージョンのプログラムに対しては Differential Slicing[5] が提案されている．この手法は，実行に成功する場合と失敗する場合の 2 つの異なる実行を観測し，出力の違いに影響を与えた動作の差分の原因を求める手法である．2 つの異なるバージョンのプログラムに対する手法としては松村らの手法 [6] がある．これは，バ

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] son.teyon.sr7@is.naist.jp

グ修正前後のプログラムに対して実行トレースを取得し，動的プログラム依存グラフと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用いて差分を検出する手法である．修正場所を比較の基点としたことで正確な依存関係の計算を達成できる一方で，動的プログラム依存グラフの構築は非常に時間がかかることがあり，メモリを大量に消費することがある．

本研究では，変更前後の Java 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 つの実行トレースに対して，依存関係の違いのかわりに，実行時の命令列の違い，変数の値の違いを比較する手法を提案する．実行トレースに対するハッシュ関数を定義して実行トレースを要約し，関数呼び出しごとの実行を表現したいハッシュ値を比較することで実行時の差分を検出する．具体的には，プログラムの変更前後の実行トレースに対して，トレース内のメソッド呼び出しを頂点，メソッドの呼び出し関係を有向辺としてマークル木を構築する．マークル木が用いるハッシュ値はそのメソッド呼び出しから終了までに記録された実行トレースを用いて計算される．したがって，あるメソッド呼び出しで異なる実行トレースが記録された時，そのメソッド呼び出しは異なるハッシュ値を持つ．最後に，異なるハッシュ値を持つメソッド呼び出しを差分としてグラフの形で出力する．

構築されたマークル木において，異なるハッシュ値が記録されたメソッド呼び出しを探索することで，異なる実行が記録されたメソッド呼び出しを列挙することができる．また，ハッシュ値の計算に用いる実行トレースの種類を指定することで，戻り値あるいは Java バイトコード実行に関する差分を検出することができる．

提案手法の有効性を調査するために，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本手法を適用し,8 個中 7 個の差分を検出できることを確認した．提案手法は，同一のデータセットを用いた既存手法と比較して，すべてのデータセットに対して少ない時間で差分を検出することができた．加えて，既存手法がメモリ不足によって計算できなかったデータセットに対して，提案手法は差分を検出した．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 章では評価，5 章では関連研究,6 章では今後の課題とまとめについて述べる．

2. 背景

本章では，背景として，ソフトウェアの変更によって生じる影響について述べる．また，本手法で用いる Omnisient Debugging の実行トレースについて述べる．

2.1 ソフトウェアの変更によって生じる影響

ソフトウェアの開発サイクルにおいて，開発者は，バグ修正や機能の追加，パフォーマンスの改善などの理由でプログラムに変更を加える．プログラムの変更は，開発者が予期しない動作や，変更によってソフトウェアに不具合を引き起こすことがある．例えば，プログラムのバグ修正の前後において，あるバグの修正がなんも関係の無い部分に影響し，異なるバグを引き起こしてしまう事がある．開発者は，プログラムの変更による影響を把握する必要がある．

開発者はプログラムの実行を比較することで，プログラムの変更によって意図しない動作が起きていないかを検証する．例えば，変更前後のプログラムに対してソフトウェアテストを実行し，変更前と変更後のテスト結果を比較することで，修正箇所の影響を検証することができる．しかしながら，テストの結果だけで影響を検証することは難しいことが知られている．ある入力において，実際にプログラムの実行が変化しているにもかかわらず，テストが偶然成功している可能性も考えられる．Yin[8] らの調査では，修正前後にテストを用いたにもかかわらず，バグ修正の 14.8 %から 24.4 %は別のバグを引き起こしていることが報告されている．また，Samim[9] らが行ったライブラリの更新を通知する bot に関する調査では，bot がライブラリの更新のために作成したプルリクエストのうち，自動テスト持つプロジェクトであってもわずか 32%のプルリクエストのみがマージされたことを報告している．Samim らは，テストの結果のみならず，API のメソッドカバレッジなどから修正前後のリスクを見積もる指標が必要であると主張している．

2.2 Omniscient Debugging

本研究では，プログラムの変更前後の実行時の差分の検出に，Omniscient Debugging で用いる実行トレースを用いる．Omniscient Debugging は，プログラムの実行中のメモリ状態を時系列順で完全に記録することで，任意の瞬間の調査を可能にする手法である．この手法では，プログラムの実行開始から終了までに実行されたすべての命令と，それによる値の変化を実行トレースとして記録する．Omniscient Debugging では，任意地点のプログラムの状態を計算機上で再現することで，命令や変数の値の観測を可能としている．これによって，開発者はプログラムを再起動することなく，ブレークポイントによるデバッグや，通常のデバッグでは困難である逆向きのステップ実行などを行うことができる．プログラムの実行開始から終了までの変数の値やスレッドの内容を観測することができるが，手法のみでバグを引き起こす原因を特定することはできない．Omniscient Debugging に用いる実行トレースには，すべての命令列が記録されているため，変更前後のプログ

```java
public static int countPrime(int a){
    int res = 0;
    for(int i = 1; i <= a; ++i){
        if(isPrime(i)){
            ++res;
        }
    }
    return res;
}
private static boolean isPrime(int a){
    if(a <= 1){
        return false;
    }
   // for(int i = 2; i * i <= a; ++i)
    for(int i = 2; i < a; ++i){
        if( a % i == 0 ){
            return false;
        }
    }
    return true;
}
```

図 1　サンプルプログラム

ラムに対して実行トレースを取得し，これを比較することで差分の検出が期待できる．しかしながら，実行トレースには膨大なデータが記録されることがあるため，2 つの実行トレースを直接比較，解析することは容易ではない．

2.3　ログ・実行トレースの比較

実行トレースやログを用いることで，開発者はプログラム実行時の詳細な情報を得ることができる．システムのログは大量に出力されることがある．

Goldstein[10] らは，システムのログから有限状態オートマトンのモデルを作成し，それらを比較することで，異常状態を検知する手法を提案した．具体的には，2 つのログが与えられたとき、k-Tails アルゴリズムを用いてそれぞれのログからモデルを構築し、そして 2 つのモデルを比較するものである．また，Amar ら [11] は、有限状態オートマトンを用いて，複数のログを比較しその差分を出力する nKDiff という手法を提案している．

3.　提案手法

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得られる実行トレースに対して，異なる実行トレースが記録されたメソッド検出手法を提案する．本研究における実行時の差分とは，異なる実行系列が記録されたメソッド呼び出しを指す．あるメソッド呼び出しにおける実行系列の変化を捉えることで，プログラムの修正によってどのメソッドに影響が波及したかを把握することができる．

提案手法の概要として，まずはじめに，各実行トレースをメソッド呼び出しごとにハッシュ値を用いて要約する．次に，このハッシュ値を用いたマークル木 [12] をそれぞれ構築する．最後に，2 つのマークル木を比較し，異なるハッシュ値を持つメソッド呼び出しを実行時の差分を表すグラフとして出力する．以降では、4 つのステップに分けて提案手法を説明する．

(1) 実行トレースの収集
(2) マークル木の構築
(3) ハッシュ値の計算
(4) 差分の出力

本研究では，Java を対象として説明を行う．以降，本稿で扱う実行トレースは Java のプログラムのものである．ハッシュ値の計算及びマークル木を用いた差分の出力の手法は，異なるプログラミング言語のトレースにも適用することができる．

表 1　実行トレースに記録される命令の例

イベント名	記録される意味
METHOD_ENTRY	メソッドの実行開始
METHOD_PARAM	メソッドの引数
METHOD_EXIT	メソッドの実行終了
LOCAL_LOAD	ローカル変数の読み出し
LOCAL_INCREMENT	ローカル変数のインクリメント

3.1　実行トレースの収集

変更前後の 2 つのプログラムに対して，同一の入力を用いてプログラムを実行し，実行開始から終了までの実行トレースをそれぞれ取得する．本研究では，Omniscient Debugging の実装の一つである松村 [13] らの手法の実行トレースを用いて差分検出を行う．この手法は，Java バイトコードの命令単位ですべてのデータの値を実行トレースとして記録する．具体的には，メソッド実行開始，実行完了におけるすべての実引数・戻り値・例外，フィールド及び配列の読み書き命令の実行における対象オブジェクト，添字，読み書きされた値を時系列順に保存する．また，オブジェクトについては，参照を区別するための ID 値を保存する．本研究では，SELogger[*1] を用いて実行トレースを取得する．SELogger は，既存の REMViwer[13] を拡張したツールであり，Java 仮想マシン内で Java エージェントとして動作する．SElogger は Java バイトコードを操作するフレームワークである ASM[*2] を用いて，解析対象となるプログラムのロードされたクラスにロギング命令を注入し実行トレースの取得する．表 1 に SELogger によって記録される命令の一部を示す．

3.2　マークル木の構築

取得した実行トレースから，マークル木 [12] を構築する．マークル木は，各頂点にハッシュ値を持つ木構造であり，葉以外の頂点は，その頂点の子ノードのハッシュ値から計

[*1] https://github.com/takashi-ishio/selogger
[*2] https://asm.ow2.io/

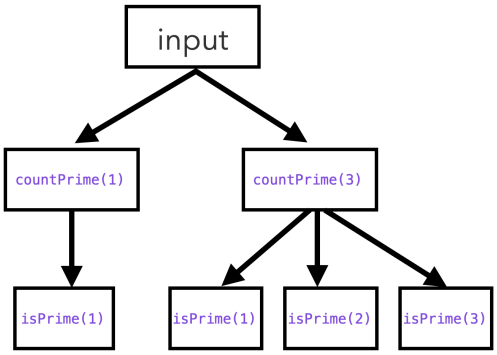

図 2 サンプルプログラムのマークル木

算される．これより，異なる 2 つのマークル木の根のハッシュ値を比較することで，木全体を効率的に比較することができる．

本手法では，メソッドの呼び出しを頂点とみなし，呼び出し関係を有向辺で表現したマークル木を構築する．例えば，実行トレース内に `method A` から `method B`, `method C` の呼び出しがこの順序で記録された時，この実行トレースから構築したマークル木は，3 つの頂点 (`method A`, `method B`, `method C`) と，2 つの有向辺 ({`method A method B`}, {`method A`,`method C`}) を持つ．手法では，メソッドの呼び出しごとに頂点とみなすため，閉路を持たず，非巡回グラフとなる．各頂点は，そのメソッド呼び出しから終了までに記録された実行トレースと，そのメソッド内でのメソッド呼び出しのハッシュ値から計算されたハッシュ値を持つ．具体的なプログラムを用いて説明を行う．1 に対するマークル木の構築を考える．サンプルプログラムは，与えられた入力以下の素数の数を返却する `countPrime` メソッドと，与えれらた入力が素数かどうかを判定する `isPrime` メソッドの 2 つからなる．`countPrime` メソッドは，引数として与えられた整数の回数だけ `isPrime` メソッドを呼び出す．入力として `1,3` を与えたときのマークル木は図 2 のようになる．

3.3 ハッシュ値の計算

本研究では，各メソッドの呼び出しごとに，そのメソッドの呼び出しから終了までの実行トレースからハッシュ値を計算する．ハッシュ値は実行トレースの要約であり，異なるハッシュ値が記録されたメソッド呼び出しは，実行時に差分が生じたものとみなす．SELogger は，メソッドの呼び出し開始に METHOD_ENTRY 命令，メソッドの終了時には METHOD_NORMAL_EXIT, または,METHOD_EXCEPTIONAL_EXIT 命令を実行トレースに記録する．あるメソッドの呼び出し命令が記録されとき，そのメソッドに対応する実行終了の命令までの実行トレースを用いてハッシュ値を計算する．本研究では，実行トレースに対して 4 つのハッシュ値を定義する．メソッド呼び出しごとに，メソッド実行開始から終了までに記録されたバイトコード命令に関するハッシュ値 (`Flow Hash`)，実行トレース内に記録されたメソッドの戻り値に関するハッシュ値 (`Param Hash`)．また，それぞれに対してマークル木の比較に用いるハッシュ値 (`Control Flow Hash` と `Control Param Hash`) を計算する．あるメソッド呼び出しにおける `Control Flow Hash`,`Control Param Hash` は，その頂点における `Param Hash` と `Flow Hash` と，呼び出し先の `Param Hash` と `Flow Hash` を用いて計算される．

Flow Hash
: 実行トレース内のあるメソッド呼び出しに対して，その実行開始から終了までに記録された Java バイトコードのイベント名から計算したハッシュ値である．ただし，メソッド内部でのメソッド呼び出しのハッシュ値はハッシュ値の計算に用いない．

Param Hash
: 実行トレース内のあるメソッド呼び出しに対して，そのメソッドの戻り値に関する実行トレースのみを用いて計算されるハッシュ値である．具体的には，METHOD_EXIT 命令が記録された時，その Value 値のみを用いて計算されるハッシュ値である．

Control Flow Hash
: Control Flow Hash は，メソッド呼び出し元 (caller) の Flow Hash と，メソッド呼び出し先 (callee) の Control Flow Hash を用いて算出される．Control FLow Hash が等しいメソッド呼び出しは，そのメソッド内で呼び出されるメソッドの実行トレースまで完全に等しいことを意味する．

Control Param Hash
: Control Param hash は，あるメソッド呼び出し時の Param Hash と，メソッド呼び出し先の Control Param hash を用いて算出される．Control Flow Hash と同じく，Control Param Hash が等しいメソッド呼び出しは，同じ戻り値が返却されることを意味する．

3.4 実行時の差分の検出

プログラム変更前後の実行トレースから構築したマークル木を比較し，異なるハッシュ値が記録されたメソッド呼び出しを差分として出力する．本研究では，実行トレースから構築したマークル木を入力として，異なるハッシュ値が記録されたメソッド呼び出しまでの実行経路をグラフとして出力する．

マークル木の性質から，あるメソッド呼び出しにおいて異なるハッシュ値が記録された時，そのメソッド呼び出し，あるいは，そのメソッドからのメソッド呼び出し先のいずれかに実行時の差分が存在する．したがって，指定したメ

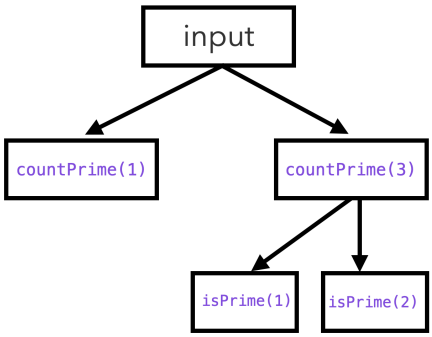

図 3　Flow ハッシュ値を用いたサンプルプログラムの変更前後の差分

ソッド呼び出しから，比較に用いるハッシュ値 (Flow Hash または Param Hash) が異なる頂点とその子供を再帰的に探索する．差分検出のアルゴリズム 1 を示す．

プログラムの変更前後で多くのメソッドが追加，削除された場合や，ある実行内でメソッドが複数回呼び出される場合，多くのメソッド呼び出しが差分として検出されることがある．本研究で用いる実行時の差分の検出アルゴリズムでは，指定されたメソッドの呼び出しから時系列順で最もはじめに差分が検出されたメソッド呼び出しのみを出力する．

サンプルプログラムの 15 行目を 14 行目に変更した場合を考える．変更前後においてプログラムの実行結果に変化はなく，`isPrime` メソッド内でのループの実行回数のみが変化する．この時，`isPrime(2)`，`isPrime(3)` にはプログラムの変更前と変更後で異なる実行トレースが記録される．したがって，比較のハッシュ値に Flow Hash を用いた時，本手法の出力は図 3 のようになる．

4. 評価実験

提案手法の有効性を評価するために，以下の観点で松村 [6] らの先行研究と比較することで評価する．

実行時間． 2 つの実行トレースを用いたマークル木の構築にかかる時間と，差分検出にかかる時間を比較する．

差分の検出能力． 変更・追加されたメソッド m が変更前後のどちらかの実行トレース内に記録されており，m の変更箇所が実行されたとき，m の呼び出しまでの実行経路を差分として出力することができるか．

4.1 実験設計

実験のデータセットとして，Defects4j[7] を用いた．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，実際に報告されたバグに対する修正前後のソースコード，バグを再現するテストケース，それらをビルドするファイル (`pom.xml` または `build.xml`) が含まれている．変更箇所はバグに関連する部分のみである．修正箇所とバグの原因は，データセット内に含まれている `defects4j info` コマンドを用いて確認した．実験の評価には，先行研究 [6] と同じく Aache Commons Lang のバグ ID 1 から 10 までのプログラムを用いた．各バグ番号の修正前と修正後のプログラムに対して，修正前後のメソッドを実行するテストを用いて実行トレースを取得した．修正前のプログラム (末尾が `b` のバグ番号を持つプログラム) に対してテストを実行した場合，テストが失敗する．失敗するテストケースが複数存在する場合は，ランダムに 1 つのテストケースを選択した．実行トレースの取得は SELogger[*3] を用いた．また，`maven` を用いたテスト実行の都合上，表中のトレースには，`JUnit` や `maven` などの実験対象とは関係がないトレースも含まれている．次に，取得した変更前後の 2 つの実行トレースから，提案手法のマークル木を構築し，それらを 2 つのハッシュ値 (`flow,param`) を用いて差分の検出を行った．提案手法によって出力された差分を表すグラフが，コードによる変更とその実行を正しく検出しているかを評価した．具体的には，1. 著者がソースコードの変更箇所を確認し，2.Defects4j のデータセットに含まれるバグ修正前後のパッチの詳細のデータ [14] を用いて，手法によって出力された差分がプログラムの変更によるものかを評価した．

Algorithm 1 実行トレースのマークル木 T_1, T_2 からの差分検出

```
1: v_0 ← root of T_1
2: u_0 ← root of T_2
3: G ← {} /* 差分を空で初期化 */
4: DepthFirstSearch(v_0, u_0, −1)
5: return G /* 差分を格納した木を返す */
6: function DepthFirstSearch(v, u, pv)
7:     G ← add v
8:     G ← add_edge {pv, v}
9:     if u, v is the same hash then
10:        return false
11:    else
12:        temrinate ← false
13:        while ¬termiante do
14:            nv ← child(v)
15:            nu ← child(u)
16:            if nv == null ∧ nu == null then
17:                return false
18:            else if nv == null ∨ nu == null then
19:                report different method call
20:                return false
21:            else
22:                terminate ←DpethFirstSearch(nv, nu, v)
23:            end if
24:        end while
25:    end if
26:    return true
27: end function
```

[*3] https://github.com/takashi-ishio/selogger/tree/b21c8318f4f9c33bdc66da8b0bc00d23a88b0d30

実行環境は，OS が CentOS Linux, メモリが 264GB,CPU が Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz であり，データセットのビルドやテストの実行は，Defects4j 内の Dockerfile から生成した Docker コンテナ内で行った．

4.2 結果

実験結果を表 2 に示す．各行はプログラムの版に対応しており，各バグ ID について，ソースコードの修正前の版を `b`, 修正後の版を `f` で表している．例えば，表中のプログラム `1b` は，Defects4j におけるバグ ID 1 の修正前のプログラムを表す．バグ ID 2 は Defects4j で非推奨のバグとなっていたため，評価から除外している．また，バグ ID 9 は，著者の環境で実行トレースを取得できなかったため，実験結果に含めていない．

提案手法は，実行トレースの記録後は、実験対象の中では最も長い実行トレースでも 7 分程度の時間でマークル木を構築することができた．既存手法は同一のトレースデータに対して動的依存グラフの構築に 2,565 秒〜5,507 秒要したと報告されており [6]，最大でも 13%程度の時間に抑えられている．また，トレースの比較については，既存手法では，バグ ID 4, バグ ID 6 の 2 つがメモリ不足によって差分の計算が完了しなかったのに対し，提案手法ではすべてのバグ ID に対して差分の計算が完了している．

表の「差分検出」と記述した列には，対応するバグに対して，Flow hash と Param hash のそれぞれを用いて手法を適用した結果，正しく差分を検出できた場合に ”○” を，差分を検出できなかった（見逃した）場合に ”X” を記載している．Flow Hash は異なる命令の実行列を，Param Hash は異なる戻り値をそれぞれ検知するため，バグの修正内容に応じて，それぞれ異なるハッシュの変化が生じる．これらのハッシュ値に基づいて検出された動的コールグラフの差分を検査したところ，8 個中 7 個のコードの変更によって生じる差分を，正しく特定することができた．特に，Flow hash を比較に用いた場合，メソッド呼び出しの追加・削除 (バグ ID 10)，代入文の削除 (バグ ID 8) や追加，if 文の削除 (バグ ID 7) などで，コードの変更による実行時の差分を正しく検出することができた．一方で，Param hash を比較に用いた場合は，代入文の削除による戻り値の変更 (バグ ID 7) などで差分を正しく検出することができた．しかし，メソッドの戻り値がオブジェクトである場合など，値の単純な等価判定が困難な場合に，差分を正しく検出することができなかった．既存手法では，バグ ID 6 は動的依存グラフの比較にメモリ不足のため失敗しており，バグ ID 8, 10 は差分なしと報告していた．これは，バグ ID 8, 10 については，命令の削除が行われるという修正であったため，前後の依存関係には影響を与えなかったためである．提案手法は，バグ ID 6 については変化を正しく検知できなかったが，バグ ID 8, 10 については，Flow Hash によって具体的な命令の実行順序としては特定のメソッドの動作に違いがあること，Param Hash によって値には変化がないことを報告していた． 開発者が動作の違いを確認するという目的では，既存手法よりも短時間で，正しく作業を実行できるようになったと言える．

両方のハッシュによる比較がうまくいったバグ ID 4 は，クラス内における異なる変数の参照と，メソッドの呼び出しの追加である．クラス内で定義されているフィールド変数である `HashMap` のキーの型が `CharSequence` から `String` へと変更と，メソッド呼び出しが追加されている．Flow hash を比較のハッシュ値として用いた場合，メソッド呼び出しの追加を差分として出力した．一方で，Param hash を比較のハッシュ値として用いた場合，手法は Flow Hash を用いた場合と異なるメソッドを実行の差分として出力した．Param hash はメソッドの戻り値に関する実行トレースのみを用いて計算されるため，戻り値に影響しない変化を捉えることができない．そのため，異なるメソッドが差分として出力されたと考えられる．Flow hash と Param hash を比較のハッシュ値として用いたときに，手法が出力する例を，図 4 と図 5 に示す．両方のハッシュがどちらもうまく機能しなかったバグ ID 6 は，メソッドの呼び出し時の引数の変更である．この変更に対して，手法は差分を正しく出力することができなかった．Flow hash は，引数が追加されても，値の変更は検知しないため，この変更を検出できなかった． また，Param hash 比較に用いた時は，変更箇所とは異なるメソッドを差分として出力した．これは，変更箇所で呼び出していた `Character.charCount` の戻り値の変化が実行に影響を与えたメソッドのみを報告したためであると考えられる．

5. 関連研究

5.1 ロギング

ロギングとは，ソフトウェアの実行の様子を観測する手段の一つであり，特定の処理の進行状況や，データやメッセージを外部に出力する手法である． 特に，プログラムの実行時の情報を記録したものを実行トレースという．Omniscient Debugging の実行トレースは，プログラムの実行開始から実行開始まですべての実行系列を保存するため，膨大なデータが記録されることがある．加えて，記録されるデータ量はプログラムの規模や実行する機能によって大きく異なるため，予め把握することが難しい．嶋利らは，限られた保存領域でプログラムの実行トレースを保存する手法 [15] と，実行トレースのインタラクティブな可視化ツール [16] を提案している． 提案手法では，プログラムの実行系列を命令ごとに一定個数記録することで，記録領域の削減とデータ及びデータの依存関係を達成している．

表 2　実験結果

プログラム番号	実行したテストケースの名称	トレースのサイズ	マークル木の構築時間	差分検出時間		差分検出	
				Flow	Param	Flow	Param
Lang 1b	NumberUtilsTest	226MB	11[s]	1434[ms]	924[ms]	○	X
Lang 1f		257MB	11[s]				
Lang 3b	NumberUtilsTest	256MB	10[s]	868[ms]	877[ms]	○	X
Lang 3f		253MB	10[s]				
Lang 4b	LookupTranslatorTest	43MB	1[s]	185[ms]	181[ms]	○	○
Lang 4f		38MB	1[s]				
Lang 5b	LocaleUtilsTest	71MB	3[s]	283[ms]	275[ms]	○	X
Lang 5f		68MB	20[s]				
Lang 6b	StringUtilsTest	2128MB	95[s]	11[s]	7[s]	X	X
Lang 6f		2109MB	93[s]				
Lang 7b	NumberUtilsTest	239MB	10[s]	896[ms]	872[ms]	○	○
Lang 7f		223MB	9 [s]				
Lang 8b	FastDatePrinterTest	130MB	5 [s]	413[ms]	523[ms]	○	X
Lang 8f		113MB	5 [s]				
Lang 10b	FastDateParserTest	9167MB	391 [s]	28[s]	29[s]	○	X
Lang 10f		7861MB	377 [s]				

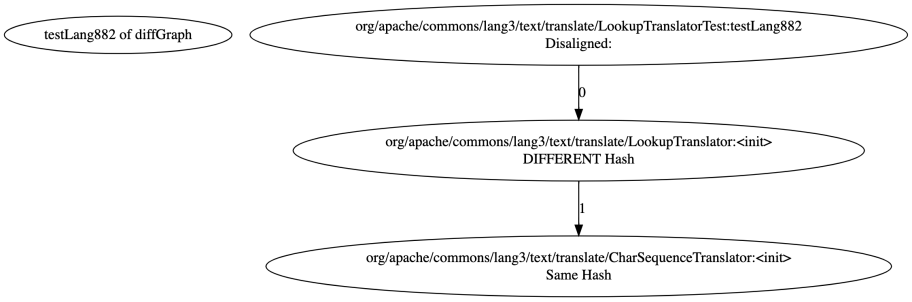

図 4　バグ ID4:Flow ハッシュで比較して検出した差分

本研究では，計算コストが高い依存関係の解析を行わず，実行トレースを用いて詳細な実行時の情報を取得し，これをハッシュ値を用いて比要約し比較している．嶋利らの実行トレースを用いることで，より高速な比較や，ハッシュ値の計算に部分的な解析を組み込むことで検出力の向上が期待できる．溝内らは，観測対象の実行が継続したロギングにおいて，動的にロギングを行う粒度を変化することでロギングによる出力を減らす手法を提案している [17]．具体的には，観測対象の状態を推定し，既知であった場合は粒度を下げ，そうでなかった場合粒度を上げてロギングを行う．また，ケーススタディにおいて，手法がデバッグに有益な情報の出力に成功している．

5.2 Change Impact Analysis

プログラムの変更時の影響を推定する技術として, Change Impact Analysis(以下，CIA) が提案されている．CIA はプログラムの変更前，変更後に適用することができ，開発者は CIA を用いることで変更によるコストの推定や変更後の影響の追跡を行うことができる．CIA の中でも，プログラムのソースコードに着目して影響を推定する手法では，静的解析と動的解析が広く用いられている [18]．Hejderup[19] らは，開発者が作成したテストに加えて静的解析を用いることで，更新時の問題を検出する手法を提案している．Huang[20] らは，オブジェクト指向言語向けの CIA の手法を提案している．メソッドとフィールドの依存関係に着目することで，高い検出精度を達成している．本手法は，メソッドの呼び出し関係を用いて差分を検出しているが，クラスなどに着目して差分を出力する手法なども考えられる．

6. まとめと今後の課題

本研究では，プログラムの変更前後における影響を把握するために，マークル木を用いて実行トレースを比較する手法を提案している．実行トレースをハッシュ値を用いて要約することで, 既存手法 [6] に比べて高速に差分を計算でき，データセットの変更前後のプログラムに対して，8 個中 7 個のコードの変更によって生じる差分を正しく出力することができた．今後の課題として，実行トレースからのハッシュ値の計算方法と，構築したマークル木の比較方法が考えられる．本研究では，2 種類のハッシュ値を定義しているが，差分の検出時にこのハッシュ値が必ずしも有用であるとは言えない．例えば，Param Hash は実行された

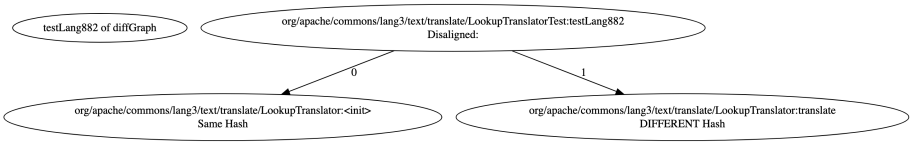

図 5 バグ ID4:Param ハッシュで比較して検出した差分

命令順序の変更などで異なるハッシュ値となってしまう．`param` ハッシュは，メソッドの戻り値に関する実行トレースのみに着目して算出しており，オブジェクトの状態を正確に表現したものではない．実行トレース内に記録された書き込み・読み込み命令に着目し，これらをハッシュ値の計算に用いることで，より正確な差分検出が可能であると考えられる．

謝辞 本研究は JSPS 科研費 No. JP18H03221 の支援を受けたものである．

参考文献

[1] Siyuan Jiang, Collin Mcmillan, and Raul Santelices. Do programmers do change impact analysis in debugging? *Empirical Softw. Engg.*, Vol. 22, No. 2, p. 631–669, apr 2017.

[2] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*. ACM Press, 2014.

[3] Bil Lewis. Debugging backwards in time. *ArXiv*, Vol. cs.SE/0310016, , 2003.

[4] David Abramson, Ian Foster, John Michalakes, and Rok Sosič. Relative debugging: a new methodology for debugging scientific applications. *Communications of the ACM*, Vol. 39, No. 11, pp. 69–77, November 1996.

[5] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Xiaodong Song. Differential slicing: Identifying causal execution differences for security applications. *2011 IEEE Symposium on Security and Privacy*, pp. 347–362, 2011.

[6] 俊徳松村, 隆石尾, 克郎井上. 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. Technical Report 8, 大阪大学大学院情報科学研究科, 大阪大学大学院情報科学研究科, 大阪大学大学院情報科学研究科, mar 2016.

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

[8] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, p. 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, p. 84–94. IEEE Press, 2017.

[10] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 282–293, 2017.

[11] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 49–59, 2018.

[12] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pp. 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[13] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎. Remviewer: 複数回実行された java メソッドの実行経路可視化ツール. コンピュータ ソフトウェア, Vol. 32, No. 3, pp. 3_137–3_148, 2015.

[14] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of SANER*, 2018.

[15] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, and Katsuro Inoue. Near-omniscient debugging for java using size-limited execution trace. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 398–401, 2019.

[16] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Naoto Ishida, and Katsuro Inoue. Nod4j: Near-omniscient debugging tool for java using size-limited execution trace. *Science of Computer Programming*, Vol. 206, p. 102630, 2021.

[17] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. Padla: A dynamic log level adapter using online phase detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 135–138, 2019.

[18] Bixin Li, Xiaobing Sun, Hareton K. N. Leung, and Sai Zhang. A survey of code ‐ based change impact analysis techniques. *Software Testing*, Vol. 23, , 2013.

[19] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? A case study of java projects. *CoRR*, Vol. abs/2109.11921, , 2021.

[20] Lulu Huang and Yeong-Tae Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *5th ACIS International Conference on Software Engineering Research, Management Applications (SERA 2007)*, pp. 374–384, 2007.