動的依存グラフの3-gramを用いた実行トレースの比較手法

ブヤンネメフ オドフー[1,a)]　石尾 隆[1,b)]　井上 克郎[1,c)]

概要：プログラムの詳しい動作を理解するために，ソフトウェアに異なる 2 つの入力を与えて実行し，その実行トレースを比較する手法が提案されている．しかし，デバッグを目的とした既存手法は，あらかじめ分析対象となる変数やデータ構造を開発者が選択する必要があり，プログラムの動作をこれから理解しようという段階で利用することは難しい．機能の実装位置の捜索やテストを目的とした既存手法は，プログラムに詳しくなくとも利用可能であるが，プログラムの機能の実装に当たる細かい動作の比較にまでは利用できない．そこで本研究では，Java プログラムの 2 つの実行トレース間の詳細な差異の一覧を計算することで，分析対象を選択しなくとも実行トレースの詳細を比較することを可能とする手法を提案する．具体的には，実行トレースを動的依存関係に従って 3 つの命令ブロックを並べた 3-gram の集合へと変換し，その集合同士を比較して差分を出力する．提案手法の評価のため，DaCapo ベンチマーク に収録された実行トレースの比較に適用するケーススタディを行い，実行トレースから小さな 3-gram の集合を抽出し，比較が可能であることを確認した．

1. はじめに

ソフトウェアの保守作業において，機能追加やデバッグ，テストを行うために，開発者はプログラムの動作を理解する必要がある．ソースコードの解析のみでは動的束縛や例外処理などの実行時情報を獲得しにくいため，プログラムに何らかの入力を与えたときに実行される命令の系列，すなわち実行トレース (Execution Trace) が重要な情報源の 1 つである [1]．ある入力に対して得られる単独の実行トレースを分析するだけでもどのように処理が実行されたかを理解することは可能であるが，異なる 2 つの入力を与えた場合や異なる実行環境で同一の入力を与えた場合といったように，異なる条件で得られる 2 つの実行トレースを比較することで，入力と処理の関係など，機能の実装を詳しく理解することが可能になる．

しかし，実行トレースの詳細を比較する既存研究は，デバッグを主な目的としており，あらかじめ分析対象となる変数やデータ構造を開発者が選択できることが前提条件となっている．そのため，プログラムの動作をこれから理解しようという段階で利用することが難しい．たとえば Differential Slicing[2] は，2 つの実行がそれぞれ異なる出力を行った場合に，その出力を行った命令を基点として実行トレースを遡り，実行が異なる原因となった入力を特定し，その入力がどのような変数や制御文を経由して出力に影響を与えたかを表すグラフ (causal difference graph) を出力する．この手法を使うためには，開発者は，分析したい機能の出力が変化するような入力をあらかじめ特定しておく必要がある．Relative Debugging[3] は，2 つの入力に対してプログラムを並行して実行し，開発者が指定したデータ構造に格納された値に差が生じたタイミングで実行を一時停止し，その情報を開発に知らせる．この手法も，あらかじめソースコードに対する理解を進め，観察すべきデータ構造を知っている必要がある．

本研究では，開発者が単一の Java プログラムから取得した 2 つの実行トレースから，その動作の差異を列挙して開発者に提示する手法を提案する．これにより，1 つの機能に異なる入力を与えて実行した場合に，どの命令で，どのような制御フロー，データフローに関する違いが生じたのかを開発者に提示する．2 つの実行トレースの比較基準として，実行された命令間の動的依存関係を用いる．ただし，動的プログラムスライシング [4] に利用されるような動的依存グラフは，実行トレースの量に比例してそのサイズが増加するため直接比較することは現実的ではない．そこで，本研究ではプログラム中のバイトコード命令をブロック単位で頂点として，それらを接続する動的依存辺の経路を抽出し，依存関係で接続された 3 つの命令ブロックの並び（3-gram）の集合へと分解する．2 つの実行トレースそ

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565-0871, Japan
[a)] b-odkhuu@ist.osaka-u.ac.jp
[b)] ishio@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

れぞれから得られた集合を比較することで，各実行トレースに固有の命令間の動的依存関係を取得し，依存関係の差異を開発者が確認できるようにする．

提案手法の有効性を検証するために，提案手法を DaCapo ベンチマーク [5] に収録された batik，fop という 2 つのアプリケーションに適用し，プログラムの動作を理解するケーススタディを行った．ベンチマークにはアプリケーションごとに small, default, large という 3 種類の実行トレースが用意されていたことから，そのうち small と default の組を比較し，プログラム実行時の出力メッセージなどから期待される差異が，実行トレースでどのような違いとして現れるかを分析した．

本論文の構成は次の通りである．まず，2 章で研究の背景について述べる．3 章で提案手法について，4 章でケーススタディについて説明する．最後に 5 章で本研究のまとめと今後の課題について述べる．

2. 背景

2.1 実行トレースの比較

実行トレースとはプログラムに何らかの入力を与えたときに実行される命令の系列である．正確な動作が分からないプログラムに対して，C 言語であれば `printf` 関数，Java であれば `System.out.println` メソッドの呼び出しを埋め込んでプログラムの実際の命令の実行順序を画面に出力し確認することは，デバッグにおける基本テクニックの 1 つである [6]．プログラム理解やデバッグにおいては，Omniscient Debugging [7] と呼ばれる，変数の実際の値の系列や条件分岐の結果が参照可能な実行トレースを用いる手法が有効であると言われている [8]．一方で，そのような詳細な実行トレースはプログラムの実行開始から終了に至るまでの経路を保存するため，一般には巨大なデータとなる．たとえば Omniscient Debugging として最初に示された実装 [7] では，2 マイクロ秒ごとにプログラムのメモリの状態のコピーを保存しておくことでプログラムの任意の時点での状態に遡ることを可能としたが，わずか 20 秒の実行で 32 ビット計算機で利用可能な 2GB のメモリを使い切ったことを報告している．このような巨大なデータから，プログラムの構造や動作を詳しく知らない状態で，開発者にとって興味のある処理の内容を分析することは難しい．

実行トレースの比較によって，興味のある処理の実行にだけ注目する手法が提案されている．Eisenbarth ら [9] は，実行トレースに含まれた機能を開発者が列挙できるとき，複数の実行トレースで実行された関数やメソッドの集合の差を，機能の差に対応づけることで，各機能と実行トレースとの対応関係を求める手法を定義した．プログラムを起動してその実行を行ってからプログラムを終了するという一連の動作を 1 つの実行トレースとして記録し，プログラムを起動してすぐにプログラムを終了するという動作を 1 つの実行トレースとして記録できれば，その差分に当たる実行トレースが目的の機能に関係するという考え方である．2 つの実行トレースに含まれる機能の対応関係が十分にある場合は，Cornelissen らの手法 [10] や Jonas らの手法 [11] で実行トレースのデータの対応関係を可視化し，実行トレースの中の特定の部分系列にだけ注目することも可能である．ただし，これらの手法では，実行トレース同士の大まかな対応関係を取ることを重視しているため，各機能の中の細かい処理のレベルでの違いを可視化結果から見分けることは難しい．

ユーザの操作履歴など，多数の実行トレースが得られる場合に，類似した実行トレースを検出する手法も提案されている．Miranskyy[12] らは現場などで集めた複数の実行トレース同士を比較し，基準となる実行トレースに類似度が高い実行トレースをフィルタリングする手法を提案している．実行トレース比較は，比較基準をいくつかの段階に分け，比較基準が粗いレベルから細かいレベルまで比較していき，似ていないものを除去していく．この手法は，複数のトレースから類似した実行トレースの組を抽出することはできるが，実行トレース同士の詳細な比較そのものを意図した技術ではない．

実行トレースのわずかな違いを区別するという観点では，出力の違いを基点としてその原因を分析する Differential Slicing[2] や，並行して実行されるプログラムに何らかの差が生じた時点を検知する Relative Debugging[3] が提案されている．これらの技術はメモリの比較や動的依存関係の比較によって詳細な比較を可能とする一方で，差異が生じた状態や原因を調べたいデータ構造や出力を開発者があらかじめ指定しなければならないため，どこに差異が生じるのかをこれから調べるという段階では利用することができない．

ソフトウェアテストの分野では，プログラムに与える入力，すなわちテストケースを変化させたとき，新しいテストケースが今までのテストケースに含まれていなかった処理を実行したかどうかをカバレッジの変化によって判断する [13]．ステートメントカバレッジは実行トレースに含まれたソースコード上の命令の集合を，ブランチカバレッジは実行トレースが通過した条件分岐の集合を，def-use カバレッジは実行トレース中に出現した変数の代入と参照の組を集合として比較する．これらのカバレッジ基準もまた，実行の差異を比較する手法と見ることができるが，本研究では依存関係の経路を使用しており，これらのカバレッジ基準よりも厳しい条件で比較を行う．

2.2 動的依存グラフ

本研究では，2 つの実行トレースから得られた動的依存関係の比較を行う．動的依存関係とは，ある命令で代入された変数の値が別の命令で使用されたというように，

プログラムの実行中に実際に発生した影響のことである．Zhang[14] らの定義によると，ある命令 s の i 回目の実行が記憶域の集合 U に格納された値を参照して D の値を定義したことを $s_i(U, D)$ で表すとすると，データ依存関係，制御依存関係はそれぞれ次のような形で与えられる．

データ依存関係 $p_i(U_p, D_p)$ と $q_j(U_q, D_q)$ において，U_q と D_p が共通要素を 1 つ以上含むとき，$p_i(U_p, D_p)$ から $q_j(U_q, D_q)$ へデータ依存関係が存在する．

制御依存関係 ある命令の実行 $p_i(U_p, D_p)$ で求められた値が $q_j(U_q, D_q)$ を実行するかどうかの条件分岐に使用されたとき，$p_i(U_p, D_p)$ から $q_j(U_q, D_q)$ へ制御依存関係が存在する．

q_j で使用した変数の値が決まった理由，q_j が実行された理由を知りたい場合には，q_j から p_i へと依存関係を遡っていくことになる [8]．本研究では，動的依存関係の経路が部分的に等しいということから，その計算の手順が部分的に等しいことを期待している．

3. 提案手法

本研究では，Java プログラムの単一のバージョンを 2 回，異なる入力で実行して得られた実行トレースを比較し，その差分を抽出する手法を提案する．具体的には，各実行トレースから得られる動的依存関係をバイトコード命令のブロックを頂点とした動的依存グラフとみなして，動的依存グラフにおける経路で命令を接続した 3-gram の集合を抽出する．2 つの実行トレースから個別に取得した 3-gram 集合の共通集合，差集合を，実行トレースにおける依存関係の差分として出力する．

プログラムでは繰り返し文の実行回数が異なる場合が非常に多いため，単純に命令の系列などをそのまま比較すると，ループの実行回数が異なる場合などに生じる命令の実行回数の差が主要な差として抽出されてしまう．たとえば，`java.io.FileOutputStream` クラスは，データ列をファイルに書き出すために，内部で 1 バイトずつデータを書き出す処理を行っている．そのため，このクラスの実行トレースを分析しようとすると，ファイルに書き出したデータ列の長さがそのまま実行トレースにおける差として現れる．このような実行トレースの差分情報は，ループ内部での処理の実行にはほとんど変化がないという点で，開発者にとって有用ではない．そこで，依存関係の部分的な経路の集合として表現することで，実行回数の厳密な差を比較に持ち込まないようにした．

提案手法は動的依存グラフが計算できれば任意の実行トレースに対して適用可能であるが，本研究の実装においては REMViewer [15] の実行トレースを使用した．この実行トレースは，Java プログラムのバイトコード単位での命令の実行系列と，ローカル変数の状態，フィールドや配列の読み書き，メソッド呼び出しによる外部システムとのやり取りの結果などを含んでおり，プログラムの実行時状態の多くを再現できる非常に詳細なものである．

```
   public static void main(String[] args) {
1:   int N = Integer.parseInt(args[0]);
2:   int sum = 0;
3:   for (int i=1; i<=N; i++) {
4:     sum = sum + i;
5:   }
6:   System.out.println(sum);
   }
```

図 1　プログラム例

```
1: $r = Integer.parseInt(args[0]);
2: int N = $r;
3: int sum = 0;
4: int i = 1;
5: if (i<=N) goto 6; else goto 9;
6: sum = sum + i;
7: i++;
8: goto 5;
9: System.out.println(sum);
```

図 2　ブロック単位に変換したプログラム

3.1 命令のブロックを単位とするプログラムの表現

Java プログラムのソースコードは，コンパイルされると多数のバイトコード命令に分解される．たとえば図 1 は，引数として渡された自然数 `N` に対して 1 から `N` までの和を出力するプログラムであるが，ソースコードとしては 6 行であっても 1 行に複数の処理が含まれている．たとえば 3 行目の `for` 文は，変数 `i` の初期化と，`i` と `N` の比較結果による条件分岐，ループを 1 周したときの `i` の値の増加という 3 つの処理から構成されている．

ソースコード上の文そのままでは命令の区切りとしては大きすぎるが，その一方でバイトコードは非常に細かい命令からなる．たとえば，図 1 の 2 行目にある変数 `sum` の初期化は，定数 0 を Java 仮想マシンのオペランドスタックにロードする命令（`ICONST_0`）と，オペランドスタック上の値をローカル変数に格納する命令（`ISTORE`）によって実現される．命令単位で動的依存関係の解析を行うことも可能であるが，定数の読み出しと格納という命令の間には条件分岐が入り込むこともなく，バイトコードだけから静的にその依存関係が確定することから，動的依存関係の単位としては小さすぎると考えられる．

本研究では，動的依存関係を求める単位として，バイトコード命令のブロックという概念を導入する．バイトコード命令のブロックは，バイトコード命令の列として表現された Java のメソッドを，以下の基準で区切って得られる命令の部分列である．

- メソッドの先頭は，ブロックの開始である．
- ある命令の終了時にオペランドスタックが空であれば，その命令はブロックの終端であり，次の命令から

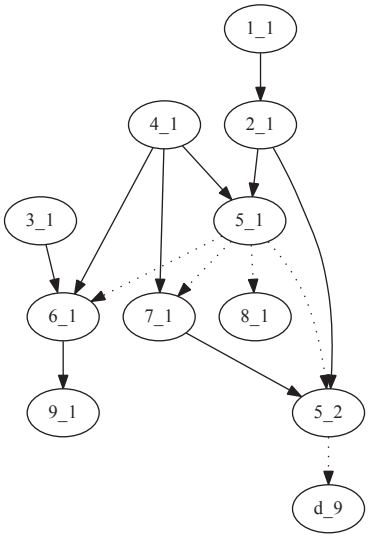

図 3 N=1 で図 1 のプログラムを実行したときのブロック間の動的依存関係

新しいブロックを開始する．

- メソッド呼び出し命令は，オペランドスタックの状態に関わらず，ブロックの終端であり，次の命令から新しいブロックを開始する．

バイトコード命令は，すべての条件分岐を IF，GOTO 命令に変換した状態となっており，ソースコードの 1 文の中でのデータのやり取りにオペランドスタックを使用している．そのため，この区切りの条件は，図 1 のプログラムを，**図 2** のような形式に変換することに相当する．図 2 の 1 行目における変数 `$r` は，オペランドスタックに格納される戻り値を表現している．この分解によって，1 ブロックの中では高々 1 回しかメソッドを呼び出さないことになり，メソッド呼び出し間の依存関係が明確にブロック間の依存関係に反映されるようになる．また，図 2 の例には含まれていないが，複数の条件を論理演算子で組み合わせた `if` 文も，個別の条件に対応する条件分岐命令を終端とするブロックの組み合わせに変換される．

3.2 ブロックを頂点とした動的依存関係

動的依存関係の概念は 2.2 節に記述した通りであるが，本研究では頂点としてブロック p の i 回目の実行 p_i を使用し，ブロックの各実行の間での依存関係として以下の関係を用いる．

- データ依存関係の定義において，D, U にはローカル変数，フィールド，配列，メソッド呼び出しの引数，戻り値を使用する．フィールドおよび配列ではオブジェクトを区別し，配列では添字の違いを区別する．つまり，同一オブジェクトの同一フィールド，あるいは同一配列の同一添字によって同一メモリ領域にアクセスしたとき，フィールドあるいは配列による動的データ依存関係があると考える．

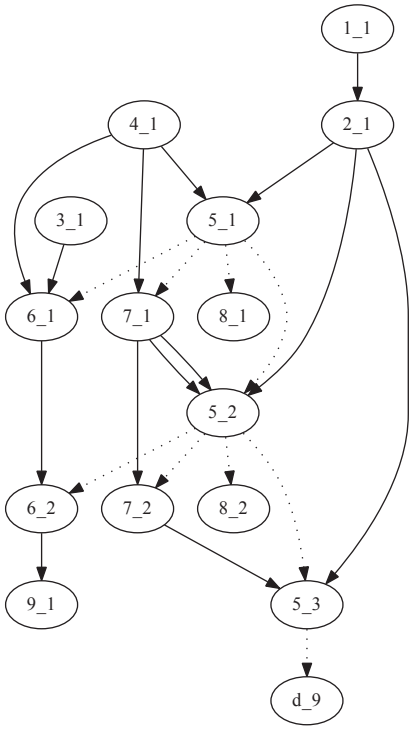

図 4 N=2 で図 1 のプログラムを実行したときのブロック間の動的依存関係

- メソッド呼び出し先が実行トレースの観測範囲外であるときは，メソッド呼び出しの引数は，戻り値に対して動的データ依存関係があると考える．
- 制御依存関係は，バイトコード上で静的に得られる依存関係と命令の実行順序を併用して求める．具体的には，ブロック p からブロック q に対して条件分岐による静的制御依存関係が存在するとき，ある q の実行 q_j は，その実行に最も近い p_i の実行に制御されているとみなす [16]．
- `else` 節を持たない `if` 文では，空の `else` 節があるものとみなし，条件分岐の結果がどうなっても，必ず 1 つ動的制御依存頂点を出現させる．

動的データ依存関係，動的制御依存関係は，$p_i \rightarrow q_j$ のように頂点間の辺として表現される．例として，図 1 のプログラムを `N`=1, 2 で実行したときに得られる動的依存関係をグラフで表現したものをそれぞれ**図 3**，**図 4** に示す．ブロックの番号は図 2 に対応しており，図中では p_i を `p_i` の形式で表記している．また，頂点 `d_9` は，条件分岐先の命令ブロック 9 が制御依存関係にないが，条件分岐の方向を表現するために導入された仮想の制御依存先頂点である．図中では実線の辺でデータ依存関係を，点線の辺で制御依存関係を示した．これは可読性のためで，この後の計算では辺の種類は区別しない．

3.3 動的依存関係による 3-gram 集合の抽出

動的依存グラフから，辺で接続された頂点 3 つの並びを

列挙した後，ブロック p の実行回数を無視して，すなわち p_i を p に変換して，3-gram の列挙を行う．直前に依存関係がない先頭の辺については，頂点がないことを示す記号 e を用いて列挙を行う．たとえば図 3 のグラフから依存関係を抽出する場合，$e \to 1_1 \to 2_1$，$1_1 \to 2_1 \to 5_1$ というように動的依存グラフ上の 3-gram を抽出した後，それを $e \to 1 \to 2$，$1 \to 2 \to 5$ というように実行回数の情報を取り除いたものに変換する．これにより，`for` 文の 1 回目の条件判定と 2 回目の条件判定（5_1, 5_2）は，いずれも入力された値 `N` を使用するが，それに関する経路は $1 \to 2 \to 5$ という同一の 3-gram になる．

3.4 3-gram 集合の比較

2 つの実行トレースからそれぞれ取得した 3-gram 集合を比較し，その共通集合，差集合を抽出する．たとえば，図 3 のグラフと図 4 のグラフから得られた 3-gram を比較した場合，変数 `sum` に関する 3-gram $4 \to 6 \to 9$ が図 3 に固有であり，一方で $4 \to 6 \to 6$，$6 \to 6 \to 9$，という 2 つの 3-gram は図 4 に固有である．

最終的な結果は，命令ブロックに割り当てた ID の 3-gram となっているが，バイトコードに記録されている命令と行番号の対応表に基づいて，命令ブロックから元になったソースコードのファイル名と行番号へと変換して閲覧することも可能である．たとえば $4 \to 6 \to 9$ というブロック間の 3-gram は，図 1 のソースコードに対応付けると以下の 3 行に対応する．

```
2: int sum = 0;
4: sum = sum + i;
6: System.out.println(sum);
```

このような差異の一覧を提示することにより，開発者は実行トレースの差異を分析することができる．

提案手法において，2 つの実行トレースから得られた 3-gram 集合が等しい場合，それらの実行トレースから得られるテストカバレッジに関して以下の性質が成り立つ．

- ブランチカバレッジが等しい．2 つの実行トレースで実行された条件分岐の集合が，そのまま動的制御依存辺の集合として 3-gram の集合に含まれるため，ブランチカバレッジが異なるなら必ず 3-gram の集合にも変化が生じる．なお，ブランチカバレッジが等しくても，条件分岐に使われるデータの依存元が異なる場合は，3-gram の集合としては異なるものとなる場合がある．
- def-use カバレッジが等しい．def-use 関係は動的データ依存辺の定義に対応するため，def-use カバレッジが異なるなら，それに対応する 3-gram の差が検出できる．def-use カバレッジが等しくても，データ依存辺の経路などが異なる場合，3-gram としては異なるものになる可能性がある．
- `for` 文などによるループ実行について，実行回数を 0 回か，1 回か，2 回以上かという 3 種類に分類したとき，ループごとの分類が等しい．これは，ループの実行継続を判定する条件分岐命令 p に対して $p \to p \to p$ という形で動的制御依存関係が発生し，3-gram に反映されるためである．

この比較は実行パス（実行された命令の列）が等しいことは保証せず，命令の実行回数が異なっていても，同じ依存関係が生じている限りは同一と判定する．たとえば図 1 のプログラムの場合，`N`=0, 1, 2, 3 までは $7_1 \to 7_2 \to 7_3$ のようにループ内部のデータ依存関係が 3-gram に反映されるため区別可能であるが，それ以上は 3-gram の長さでは表現できないため，機能の実行回数がそれ以上増えても 3-gram の集合は変化せず，区別することができない．そのかわり，ループの実行回数によらず，計算の差異や条件分岐の差異が生じたかどうかだけに注目することができる．

4. ケーススタディ

提案手法の有効性を検証するために，同一プログラムの異なる入力に対する実行トレースを提案手法によって比較し，プログラム動作理解のケーススタディを行った．ケーススタディの対象は，DaCapo ベンチマーク [5] に収録された 2 つの Java アプリケーション batik, fop である．DaCapo ベンチマークは，アプリケーションごとに，small, default，large という 3 つの実行オプションを提供している．それぞれが実行する機能について明確な情報が与えられているわけではないが，small よりは default が，default よりは large のほうが多くの機能を実行し，標準出力にメッセージを出力する．batik, fop について，ケーススタディ開始時点で各アプリケーションに対して持っていた我々の事前知識は以下の通りである．

batik

Apache Batik は，Scalable Vector Graphics 画像を取り扱うソフトウェアである．DaCapo ベンチマークにおける batik の実行は，拡張子 svg の画像ファイルを入力として，それを拡張子 png の画像ファイルに変換する機能を実行する．実行オプションとして small が選ばれた場合は，単一のファイル mapWaadt.svg の変換を行い，default が選ばれた場合は，mapWaadt.svg, mapSpain.svg, sydney.svg という 3 つのファイルの変換を行うことが標準出力に書き出される．

fop

Apache FOP は XSL formatting objects (XSL-FO) と呼ばれる入力ファイルを解析し，画像やテキストなど様々な形式でその内容を可視化するレンダリングソフトウェアである．DaCapo ベンチマークにおける fop の実行は，pdf ファイルを作成する．標準出力には何もメッセージを出力しないため，small と default

表 2　各実行トレースに固有・共通の 3-gram の個数

実行トレース名	固有	共通
batik small	405	14,413
batik default	7,897	
fop small	1,256	10,676
fop default	4,959	

で標準出力から明確な動作の違いを認識することはできない．

プログラムの実行を記録し，Java のバイトコード命令の実行順序，Java 仮想マシンのローカル変数の状態やオペランドスタックの状態，ヒープメモリの状態などを再現できる実行トレースを記録した．そして，small と default という 2 つの実行トレースをそれぞれ提案手法によって 3-gram の集合へと変換した．表 1 に，ケーススタディの対象とした各実行トレースの本来の実行時間と，その実行動作をすべて記録した実行トレースのファイルの総容量，そこから抽出される 3-gram の集合のファイル容量，そして 3-gram 集合の要素数を示す．表 1 から，ファイルに保存すると GB 単位の情報を含む巨大な実行トレースに対しても，動的依存グラフを 3-gram の集合として表現することよって，実行トレースの動作を非常に少ない要素数で表現できたことがわかる．例えば，batik の default の実行トレースの場合，3,131 個のメソッドがのべ 1700 万回ほど実行されるが，その動作を 3-gram 集合で表すと 470KB 程度のファイルとして保存することが可能であった．また，batik, fop の両方で，実行トレースのサイズの差に比べて，3-gram 集合の要素数の増加量は限られている．これは，batik の場合はファイルの変換個数が異なるといったように，ベンチマークの実行トレースが同じ種類の機能の実行規模を変更するという形で設計されているためだと考えられる．

4.1　実行トレースの比較結果

表 2 には，3-gram 集合を比較し，各実行トレースに固有の 3-gram と，2 つの実行トレースに共通の 3-gram に分類した結果を示す．ここから，各アプリケーションの実行トレースが依存関係の多くを共有しており，small で実行された機能の多くが default にも含まれていることが読み取れる．

得られた 3-gram の集合の差から batik small と default の動作の違いが読み取れた情報の例として，ソースコードの抜粋を**図 5** に示す．図中の行番号は，3-gram に関係する場所にのみ示している．batik は指定された SVG ファイルに対する変換処理を実行するが，default の実行トレースに固有の 3-gram として 677 → 679 → 684 という経路が存在し，677 行目で代入された `sources` が 679 行目の `if` 文の条件節の 1 つ目の条件 `sources.size()==1` で使用さ

```
[SVGConverter.java]
     public void setDst(File dst) {
492:   this.dst = dst;
     }

     public void execute() throws
                          SVGConverterException {
677:   List sources = computeSources();
       List destFiles = null;
679:   if (sources.size() == 1 &&
           dst != null && isFile(dst)) {
         dstFiles = new ArrayList();
         dstFiles.add(dst);
       }
       else {
684:     dstFiles = computeDstFiles(sources);
       }
```

図 5　batik small と default で検出された動作の差異

れ，それが「変換元ファイルが 1 つではない」と判定されて `else` 節の文の実行につながったと読み取れる．small の実行トレースには 492 → 679 → 684 という 3-gram が存在しており，実行トレースを確認したところ，こちらは変換対象のファイルがただ 1 つなので出力対象がファイルかどうかを判定する 3 つ目の条件文まで進み，その後 `else` に進んでいた．この処理の違いは，batik は単一ファイルを変換するときは出力先としてファイル名とディレクトリ名の好きな方を利用できるが，複数ファイルを変換するときの出力先の指定は必ずディレクトリとなる，という機能の実装が影響したものだと考えられる．

図 5 の 684 行目に書かれた `computeDstFiles` というメソッドの呼び出し先では，`sources` に格納されたファイルを順に処理しているので，ループの実行回数が 3-gram の差として検出されたが，それ以外の依存関係は small と batik で等しいことから，ファイルごとの処理には特に変化がないと推測できた．

このように，small と default では処理の実行回数に差があっても，その差が影響した範囲だけに注目して調査を行えるようになる点が，提案手法の特徴であるといえる．なお，batik default 固有の 3-gram には，small では実行されていない多数のクラス，メソッドが含まれていたことから，default は small と同じ動作を 3 回繰り返すだけではなく，さらに追加の処理を実行していると判断することが可能であった．一方で，新しいメソッドが 1 つ実行されただけでも，そのメソッドに関する多数の 3-gram が差分として抽出されるというのが提案手法の弱点である．今後提案手法を活用していくためには，そのようなトレース固有の 3-gram をクラスやメソッド単位で整理して閲覧するためのビューアの整備が必要であると考えられる．

表 1　各実行トレースごとの実行時間とファイルのサイズ

実行トレース名	実行時間	実行トレースファイルのサイズ	3-gram ファイルのサイズ	3-gram 集合の要素数
batik small	2,542ms	7.39GB	304KB	14,818
batik default	4,430ms	40.2GB	470KB	22,310
fop small	1,747ms	1.53GB	253KB	11,932
fop default	2,886ms	8.56GB	336KB	15,635

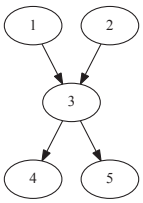

図 6　2-gram と 3-gram で差が生じる動的依存関係の例

表 3　3-gram を使わないと依存関係の経路が認識できなかったブロックの数

実行トレース名	命令ブロック数
batik small	191
batik default	296
fop small	161
fop default	171

```
[SVGSVGElementBridge.java]
     public GraphicsNode createGraphicsNode(...) {
       CanvasGraphicsNode cgn;
       cgn = (CanvasGraphicsNode)
             instantiateGraphicsNode();
160:   cgn.setViewingTransform(viewingTransform);

[SVGAbstractTranscoder.java]
     protected void transcode(Document document, ...) {
       CanvasGraphicsNode cgn =
         getCanvasGraphicsNode(gvtRoot);
287:   cgn.setViewingTransform(Px);

[CanvasGraphicsNode.java]
     public void setViewingTransform(
                                AffineTransform at) {
115:   this.viewingTransform = at;
116:   if (positionTransform != null) {
117:     transform = new AffineTransform(
                          positionTransform);
118:     if (viewingTransform != null)
119:       transform.concatenate(viewingTransform);
120:   } else if (viewingTransform != null)
121:     transform = new AffineTransform(
                          viewingTransform);
```

図 7　2-gram では表現できない経路に対応する batik のソースコード（抜粋）

4.2　3-gram と 2-gram の比較

本研究では 3-gram の集合を比較対象に使用したが，2-gram，つまり動的依存辺の集合を単純に比較するだけでも，ブランチカバレッジや def-use カバレッジが等しいことを確認することができる．そこで，2-gram の集合による比較では検出することができないような依存関係の経路が実行トレースに含まれているかどうかを調査した．

3-gram では表現されるが 2-gram による表現ができない経路の構造を**図 6** に示す．2-gram では 4 本の辺の集合として与えられるため，たとえば $1 \to 3 \to 4$，$2 \to 3 \to 5$ という 2 つの経路が既に存在するときは，新たに $2 \to 3 \to 4$ という 3-gram が出現したとしても，2-gram の集合では認識することができない．これは，あるブロック B（図 6 中での頂点 3）から見て，そこに依存辺をもつブロックが x 個，B からの依存辺があるブロックが y 個あるとすると，実行経路となる 3-gram が最大で xy 個存在するのに対し，2-gram では隣接したブロック間の $x + y$ 通りの関係までしか取り扱えないためである．このような 3-gram 集合でなければ存在を認識できないような実行経路を持つ命令ブロックの数を各実行トレースから検出したところ，**表 3** のような結果が得られた．ここでの個数は，図 6 におけるブロック 3 に相当する命令ブロックの数である．この結果から，3-gram の利用が，2-gram よりも実行トレースのより正確な表現として意味があることが確認できる．また，batik の比較では 20 個，fop では 14 個の 3-gram が，2-gram を使っての比較では見逃されることを確認した．実際の例としては，次のような 3-gram の組が挙げられる．

```
SVGSVGElementBridge.java:160
  -> CanvasGraphicsNode.java:115
  -> CanvasGraphicsNode.java:119
SVGAbstractTranscoder.java:287
  -> CanvasGraphicsNode.java:115
  -> CanvasGraphicsNode.java:121
```

この 2 つの 3-gram に対応するソースファイル 3 つの断片を**図 7** に示す．図中では，依存関係に関わるコード以外は省略した．これらの 3-gram は，いずれも先頭の要素がメソッド呼び出し元となっており，`setViewingTransform` メソッドの内部でオブジェクトの状態の差により条件分岐が起きた結果を表現している．2-gram では，2 つの呼び出し元のどちらからのデータフローが `if` 文のどちらの経路にたどり着いたか表現されないが，3-gram を用いると依存関係の経路を知ることが容易である．N-gram の N を大きくしていくとさらに経路の識別能力が向上することは期待されるが，そのコストと効果のトレードオフの評価は今後の課題である．

5. まとめ

本研究では，Java プログラムの 2 つの実行トレースを比較し，共通する依存関係と各トレースに固有の依存関係を求める手法を提案した．提案手法では，動的依存グラフ全体を比較するかわりに，動的依存グラフを命令のブロック単位を頂点とした 3-gram の集合に変換して比較を行っている．動的依存グラフの辺（2-gram）の集合でブランチカバレッジや def-use カバレッジなどの同一性を保証できるが，3-gram はさらに詳細な比較となっている．

提案手法の有効性を検証するために，プログラム動作理解のケーススタディを行った．DaCapo ベンチマークに収録された 2 つの Java アプリケーション batik, fop の実行トレースに対して提案手法を適用し，得られた差分の事例を確認した．その結果，実行トレースの 3-gram 集合は実行トレースに比べて非常に小さなデータ量で表現することができること，同一の処理が複数回実行されているような場合の差異を無視できることを確認し，また，3-gram による比較が有効に働くと思われる事例も数多く実行トレースに含まれていたことを確認した．

3-gram は小さな差異を抽出できるが，あるメソッドが一方の実行トレースでは実行されるが他方では実行されないという場合，そのメソッドに関する動的依存関係から多数の 3-gram が差分として抽出されてしまう．そのため，クラスやメソッド単位で 3-gram の差分を整理して閲覧できるようなビューアを整備し，開発者が実行トレースを分析する作業に適用することが今後の課題である．また，本研究では 3-gram を適用することでループの回数を 0 回，1 回，複数回とを区別することや，データ依存関係の短い経路を区別することを可能としたが，一般に N-gram の N を大きくするとさらに経路の識別能力が大きくなることが期待できることから，そのコストと効果のトレードオフの評価もまた今後の課題である．

謝辞　本研究は科研費 No.26280021 の助成を得た．

参考文献

[1] Zaidman, A.: Scalability solutions for program comprehension through dynamic analysis, PhD Thesis, University of Antwerp (2006).

[2] Johnson, N. M., Caballero, J., Chen, K. Z., McCamant, S., Poosankam, P., Reynaud, D. and Song, D.: Differential slicing: Identifying causal execution differences for security applications, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 347–362 (2011).

[3] Abramson, D., Chu, C., Kurniawan, D. and Searle, A.: Relative debugging in an integrated development environment, *Software Practice and Experience*, Vol. 39, No. 14, pp. 1157–1183 (2009).

[4] Agrawal, H. and Horgan, J. R.: Dynamic program slicing, *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 246–256 (1990).

[5] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D. and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190 (online), DOI: http://doi.acm.org/10.1145/1167473.1167488 (2006).

[6] Zeller, A.: *Why Programs Fail - A Guide to Systematic Debugging, the Second Edition*, Morgan Kaufmann (2011).

[7] Lewis, B.: Debugging Backwards in Time, *Proceedings of the 5th International Workshop on Automated Debugging*, (online), available from ⟨http://arxiv.org/html/cs/0309027v4⟩ (2003).

[8] Ko, A. and Myers, B.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *Proceedings of the 30th International Conference on Software Engineering*, pp. 301–310 (2008).

[9] Eisenbarth, T., Koschke, R. and Simon, D.: Locating features in source code, *IEEE Transactions on Software Engineering*, Vol. 29, No. 3, pp. 210–224 (2003).

[10] Cornelissen, B. and Moonen, L.: Visualizing similarities in execution traces, *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pp. 6–10 (2007).

[11] Trumper, J., Dollner, J. and Telea, A.: Multiscale visual comparison of execution traces, *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pp. 53–62 (2013).

[12] Miranskyy, A. V., Madhavji, N. H., Gittens, M. S., Davison, M., Wilding, M. and Godwin, D.: An iterative, multi-level, and scalable approach to comparing execution traces, *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pp. 537–540 (2007).

[13] Zhu, H., Hall, P. a. V. and May, J. H. R.: Software unit test coverage and adequacy, *ACM Computing Surveys*, Vol. 29, No. 4, pp. 366–427 (online), DOI: 10.1145/267580.267590 (1997).

[14] Zhang, X., Tallam, S. and Gupta, R.: Dynamic slicing long running programs through execution fast forwarding, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 81–91 (2006).

[15] Matsumura, T., Ishio, T., Kashima, Y. and Inoue, K.: Repeatedly-Executed-Method Viewer for EfficientVisualization of Execution Paths and States in Java, *Proceedings of the 22nd International Conference on Program Comprehension* (2014).

[16] Xin, B. and Zhang, X.: Efficient Online Detection of Dynamic Control Dependence, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 185–195 (2007).