修士学位論文

題目

動的依存グラフの 3-gram の比較によるプログラム動作理解支援

指導教員

井上 克郎 教授

報告者

Buyannemekh Odkhuu

平成 27 年 2 月 6 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 26 年度 修士学位論文

動的依存グラフの 3-gram の比較によるプログラム動作理解支援

Buyannemekh Odkhuu

内容梗概

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．プログラムの動作を理解する手法の 1 つとして，プログラムに異なる入力を与えて実行したときの命令の系列，すなわち実行トレースを比較することで，プログラムの動作の変化を分析する方法が知られている．そのような手法の 1 つである Differential Slicing は，開発者が指定した出力命令を基点として 2 つの実行トレースを遡り，与えられた入力が変数や制御文を経由して出力に影響するまでの依存関係の経路の差を計算する．このような手法は詳細なプログラムの動作の分析に有効であるが，プログラムの各機能の入出力を知っている開発者にのみ利用できる技術であるため，機能について詳しく理解していない状態の開発者は利用することができない．本研究では，開発者がプログラムに異なる入力を与えた場合に，どの命令で，どのような動作の違いが生じたのか，その差分を列挙する手法を提案する．2 つの実行トレースを命令間の動的依存関係によって比較するという点は既存手法と同様であるが，動的依存関係のグラフ全体を比較するのではなく，プログラム中の命令間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 つの命令の並び（3-gram）の集合として実行トレースを表現し，集合間の比較によって各実行トレースに固有の動的依存関係を列挙する．実験では，提案手法の有効性を検証するために，DaCapo ベンチマークに収録された batik，fop という 2 つのアプリケーションから 2 つずつ取得した実行トレースを比較し，その実行トレースに含まれる動作の差異を理解するケーススタディを行った．その結果，大規模な実行トレースに対しても小さな 3-gram 集合が得られること，条件分岐の違いのようなわずかな動作の違いを抽出できること，動的依存関係を 3-gram にしたことで検出できるような動作が実行トレースに多数含まれることを確認した．

主な用語

実行トレース比較
動的依存グラフ

依存関係

目 次

1 はじめに **4**

2 背景 **6**

2.1 実行トレースとは 6

2.2 実行トレースの比較 6

2.3 動的依存グラフ 8

3 提案手法とその実装 **9**

3.1 提案手法における動的依存グラフ 9

3.2 動的依存グラフの分解 10

3.2.1 ブロックの定義 16

3.2.2 ブロックを頂点とした 3-gram 表現 18

3.2.3 動的依存グラフの分解の実装 18

3.3 分解した動的依存グラフの比較 23

3.4 提案手法の実装 25

3.4.1 ブロックを頂点に持つグラフの取得 25

3.4.2 ブロックを頂点に持つグラフの解析 26

4 ケーススタディ **28**

5 まとめと今後の課題 **33**

謝辞 **34**

参考文献 **35**

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．しかし，長期間に渡って継続的に保守されてきたシステムでは，保守を担当する人がシステムについての知識を十分には持っていないことが多い．また，システムに関する設計文書やユーザ用の説明書も機能の変更を反映しておらず，そこから正確な情報が得られないことがある．

プログラムの動作を理解するための技術として，プログラムの実行トレースの比較が挙げられる．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り，動的束縛や例外処理など，ソースコードの静的解析では得ることが難しい重要な実行時情報を含んでいる [13]．実行トレースはプログラムの動作を正確に表すので，実行トレースを比較することでプログラムの動作を比較できる．保守作業において，開発者が作業対象の機能についての理解を深めたい場合，次のような実行トレースの比較が考えられる．

(1)　ある機能を実行した場合とその機能を実行しなかった場合で，実行された命令の差分から，機能の実現に関する命令が書かれたソースコードを特定する [6]．

(2)　わずかに異なる 2 つの入力を与えて同一の機能を実行し，入力の違いによって生じる動作の違いを調査することで，その機能の詳細な動作を理解する．

このうち (1) は Feature Location と呼ばれる技術であり，既に様々な研究がなされている [5]．(2) については，デバッグ支援という観点から，Differential Slicing，Relative Debugging という 2 つの技術が提案されている．

Differential slicing[7] 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) を出力する手法である．この手法を使う場合，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[1] は，対話的デバッガの一種であり，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が異なる値を持ったタイミングで実行を一時停止し，開発者に知らせる手法である．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．

これらの技術は，入力の違いが機能の出力や計算途中のデータ構造に影響を与えることを知っている場合にのみ利用できる技術であり，機能について詳しく理解していない段階では利用することができない．そこで本研究では，開発者が機能を実装したソースコードを読解している段階で，その機能に異なる入力を与えた場合に，どの命令で，どのような動作の違

いが生じたのかを抽出する手法を提案する．本手法により，実行トレース全体を比較した結果の差分を列挙することで，Differential slicing や Relative debugging による詳細な調査の開始が可能となる．

本研究では，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．動的プログラムスライシング技術 [2] を用いると，1 つの実行トレースから動的依存グラフを計算することができるが，このグラフは実行トレースの量に比例してサイズが増加するので，それを直接比較することは現実的ではない．そこで，プログラム中の命令を頂点として，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 つの命令の並び（3-gram）の集合へと分解する．2 つの実行トレースそ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が得られる．

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 [3] 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, default とを比較し，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 章では，提案手法，その実装について説明する．4 章では，ケーススタディを説明し，最後に 5 章では本研究のまとめと今後の課題について述べる．

2　背景

本章では，背景として，実行トレースについて述べ，実行トレースを比較する様々な既存技術の状況について述べる．

2.1　実行トレースとは

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．プログラムの実行の制御は，動的束縛や例外処理など，実行時のデータに応じて変化するため，実行トレースには静的解析では得られない有用な情報が含まれている [13]．

プログラム理解に実行トレースを応用する方法の 1 つが，得られた実行トレース自体を用いて，プログラムの動作の概要あるいは設計図などを抽出する方法である．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制御の流れを文字列の形で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 つとして知られている [14]．プログラムの実行が長くなるとそのような情報の系列であっても膨大な量に達する可能性もあるが，その概要を抽出する可視化技術も存在する．たとえば谷口ら [17] は，Java プログラムの実行から得られるメソッド呼び出しの系列の中から，繰り返し実行されたメソッド呼び出しの部分系列を認識し，繰り返し回数の注釈が付いたシーケンス図形式で可視化する手法を提案している．

実行トレースとしては，単純な行番号の系列やメソッド呼び出しの系列などが頻繁に使用されるが，解析の目的によってはより詳細な情報も使用される．プログラム理解やデバッグにおいては，Omniscient Debugging [9] 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とする手法が有用であると言われている [8]．本研究では，Omniscient Debugger の一種である REMViewer [10] の実行トレースを使用するが，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．

2.2　実行トレースの比較

実行トレースにはプログラムの実行開始から終了に至るまでの経路が保存されるため，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 [9] 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．

Eisenbarth ら [6] 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 [4] の手法や Jonas ら [12] の手法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は，類似した実行トレースを取り除くことも行われる．Miranskyy[11] 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．この手法では，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， Relative Debugging[1] が提案されている．この技術は，対話的デバッガとして，2 つの並行して実行されるプログラムに違いが生じた時点を検知する．Differential Slicing[7] は，出力の違いを基点として，プログラムの制御フローの違いと，値の違いがどのように入力から伝播してきたか，その依存関係を解析し，開発者に提供する．これらの技術は，あらかじめ開発者が注目した差分を指定しなければならないため，機能について詳しく理解していない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ることで，プログラムに様々な処理を実行させ，正しく動作するかを検査する．その中で，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判断する基準が，カバレッジである [16]．ステートメントカバレッジは，テストケースを実行したとき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比較する．ブランチカバレッジは，実行中に通過した条件分岐の集合を比較する．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．既に得られた実行トレース集合から得られたカバレッジと，新しいテストケースに対する実行トレースからカバ

レッジを比較することで，新しいテストケースに含まれる新しい処理を識別することができる．ただし，カバレッジの比較基準によっては，プログラムのあらゆる動作を網羅しなくともテストが完了した（これ以上新しい要素が増えない）状態に到達しうる．

2.3 動的依存グラフ

プログラムの実行では，ある命令で代入された変数の値が別の命令で使用されるというように，命令間に依存関係が存在する．実行された命令を頂点とし，依存関係を辺として接続していくと，動的依存グラフが構成できる．Zhang[15] らが定義したプログラム実行全体における動的依存グラフ $DDG(N, E)$ は，頂点の集合 N と有向辺の集合 E からなるグラフであり，$n_i \in N$ である各頂点 n_i は命令 i の n 回目の実行を表し，$m_j \to n_i \in E$ はであるエッジは命令 m の j 回目の実行から命令 n の i 回目の実行へのデータ依存関係，制御依存関係，潜在依存関係（potential dependence）のいずれかを示す．ある命令 s の j 回目の実行に対して，参照された変数（値）を U，定義された変数（値）を D として，その実行を $s_j(U, D)$ で表すとすると，依存関係はそれぞれ次のような形で定義される．

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 が重なる場合（共通要素を含む場合す）$p_i(U_p, D_p)$ 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 p の実行 $p_i(U_p, D_p)$ が，$q_j(U_q, D_q)$ の D_q に含まれる変数に制御されている（依存している）場合， $q_j(U_q, D_q)$ から $p_i(U_p, D_p)$ へ制御依存関係が存在する．(ほとんどの場合 q は条件節に対する命令である)

潜在依存関係 ある条件節の命令 p 実行 d が決める経路によって，$s_j(U, D)$ での D が変わる場合，d から $s_j(U, D)$ へ潜在依存関係が存在する．

頂点 p から q にデータ依存関係があるとき，p は，q で使用された変数を計算した命令の実行である．頂点 p から q に制御依存関係があるとき，p は，q を実行することを決定した条件分岐命令の実行である．これらのことから，q で使用した変数の値が決まった理由，q が実行された理由を知りたい場合には，q から p へと依存関係を遡っていくことになる [8]．動的プログラムスライシング（Dynamic Program Slicing）[2] では，1 つの動的依存グラフにおいて，ある実行時点での変数の値に影響を与えた命令列を特定するため，基点となる 1 頂点からグラフを遡る．Differential Slicing は，2 つの実行トレースから得られた動的依存グラフの比較を行う [7]．本研究では，データ依存関係，制御依存関係に関する頂点間の 3-gram を抽出し，その集合の比較を行う．

3　提案手法とその実装

本研究では，Java プログラムを対象に，プログラムの実行動作を表す動的依存グラフを分解し，重複するものを取り除いて，比較する手法を提案する．本手法は比較する 2 つの実行トレースとプログラムのソースコードが与えられたとき，各実行トレースから得られる動的依存グラフからブロック処理を特定し，ブロック処理間の依存関係単位で動的依存グラフを 3-gram 分解し，集合にする．そして，各実行トレースおける 3-gram 分解した集合を比較し，その差分と共通するものを出力する．

本章では，まず，本研究で定義する動的依存グラフ，ブロック処理，その必要性について説明する．次に，提案手法の流れを，動的依存グラフの分解と分解した動的依存グラフ比較に分けて説明する．

3.1　提案手法における動的依存グラフ

Java プログラムの実行はそのソースコードが表現するバイトコード命令の実行によって実現される．本研究ではバイトコード命令の位置を区別するためにユニークな番号（以降，命令番号と呼ぶ）を付けた．また，以降ではバイトコード命令のことを単に命令と呼ぶ．さらに，プログラム実行開始からある命令 i の N 回目の実行を i_N と記述する．i_N はその命令が実行される位置情報（クラス名，メソッド名，行番号，命令番号）と時刻情報を持つ．

本研究における動的依存グラフとは，プログラムのある実行で，実行された命令（あるいは命令が実行される時刻）を表す頂点と，命令間の依存関係を表す辺をもつ有向グラフである．（任意の命令の異なる実行において，異なる頂点を持つ．）

命令間の依存関係は，プログラム実行開始からある命令 i の N 回目の実行 i_N は他の命令 j の M 回目の実行 j_M にから影響を受ける場合に発生する．この場合，j_M から i_N への命令間の依存関係が存在し，$Dep(j_M, i_N)$ と書く．j_M が実行される時刻を $t1$，i_N が実行される時刻を $t2$ とすると，$Dep(j_M, i_N)$ を $Dep(t1, t2)$ とも書く．命令間の依存関係にはデータ依存関係（$Data$）と制御依存関係（$Control$）の 2 種類がある．それぞれの定義を以下に述べる．

データ依存関係

変数による依存関係　プログラム実行時のある時刻 $t1$ において，命令 p の N 回目の実行 p_N が識別子 v の値を定義したとする．それ以降，識別子 v の値が上書きされることなく，時刻 $t2$ において，命令 q の N 回目の実行 q_M で使用されたとき，p_N から q_M へのデータ依存関係が存在し，$Data(p_N, q_M)$ が成り立つものとする．こ

の場合，$Data(p_N, q_M)$ を $Data(t1, t2)$ とも書ける．識別子として，フィールド，配列，ローカル変数のどれもありうる．ただし，フィールドと配列において，オブジェクトが同じものである．

オペランドスタックによる依存間関係 プログラム実行時のある時刻 $t3$ において，命令 r の N 回目の実行 r_N がオペランドスタックに値 l をプッシュしたとする．それ以降，時刻 $t4$ において，命令 s の M 回目の実行 s_M で値 l をオペランドスタックからポップし，参照したとき，r_N から s_M へのデータ依存関係が存在し，$Data(r_M, s_N)$ が成り立つものとする．この場合，$Data(r_M, s_N)$ を $Data(t4, t3)$ とも書ける．

制御依存関係

命令 s が条件分岐命令 p に静的に依存しているとき，s_N は直前の命令 p の実行 p_M に依存する．この場合，p_M から s_N へ制御依存関係が存在し，$Control(p_M, s_N)$ が成り立つものとする．p_M の実行時刻を $t1$ とすると $Control(p_M, s_N)$ を $Control(t1, t2)$ をとも書ける．

Java バイトコードの条件分岐命令とは `if_acmp,if_icmp,if,ifnonnull,ifnull` のことを指す．また，else がない if 文において，ダミーの分岐先を生成し，条件分岐命令の分岐先が if 文の制御する範囲外の場合，ダミーの分岐先を通ったことにし，条件分岐命令からダミー分岐先へ制御依存を引く．

3.2 動的依存グラフの分解

Java プログラムの動作を細かく分解すると，実行される命令の集合とそれの間の依存関係（動的依存グラフ）で表現できる．Java プログラムの動作をバイトコード命令より荒い単位で表すこともできる．例えば，分解単位を行とすると，実行された行とそれらの間の依存関係でプログラムの動作を表現できる．そのため，実行トレース比較の基準を命令にし，動的依存グラフを比較すれば，実行トレース比較によって動作を正しく比較できる．

しかし，動的依存グラフのサイズは，実行トレースに含まれる命令数に比例して増加する．また，実行トレースのサイズは短時間の実行でも膨大となる．例えば，dacapo ベンチマークソフトの 1 つで，拡張子 png の画像ファイルを拡張子 svg ファイルに変換する batik の 2014 ミリ秒間の実行で実行された命令がおよそ $3.3*10^9$ 個あり，実行トレースサイズが 7.39GB になった．そのため，動的依存グラフを直接比較することは現実的ではない．そこで，本研究では命令間の依存関係によって表される動作の細かい表現をなるべく落とさずに動的依存グラフを分解して，グラフ上での部分的な経路の集合として表現し，重複するものを除去して，サイズを小さくしてから比較を行う．

命令間の依存関係を表す動的依存グラフの分解する単位の長さによってその動作表現が変わる．分解単位が短すぎる場合は処理と処理間の依存関係がわからなくなる．分解単位が長いほど詳細に動作表現ができるが，意味のある区切りで動的依存グラフを分解できているとは限らない．

動的依存グラフの分解単位が短すぎる場合：命令間の依存関係の長さが 2 の場合（依存関係で接続された 3 つ命令の並び）を考える．ある命令において，その命令に影響を与えた命令と，その命令が影響を与える命令の依存関係は保つことができる．しかし，動的依存グラフを分解後に，各分解した単位が表現できる実行経路の長さは小さくなり，プログラム実行における詳細動作表現ができなくなる．そのことについて，図 1 に示す Java プログラムのコード片において，ソースコード上で 13 行から 18 行までの処理を例に説明する．

```
 9       a=5;
10       b=7;
11       c=10;
12
13       d=flag1?a+c:b+c;
14
15       if(flag2){
16         sum=d;
17       }else{
18         sum=d+c;
19       }
```

```
(line=9)
ICONST_5
ISTORE 1 (a)
(L00012)
(line=10)
BIPUSH
ISTORE 2 (b)
(L00016)
(line=11)
BIPUSH
ISTORE 3 (c)
(L00020)
(line=13)
ILOAD 6 (flag1)
IFEQ L00028
ILOAD 1 (a)
ILOAD 3 (c)
IADD
GOTO L00033
(L00028)
FRAME-OP(0)
ILOAD 2 (b)
ILOAD 3 (c)
IADD
(L00033)
FRAME-OP(4)
ISTORE 4 (d)
(L00036)
(line=15)
ILOAD 7 (flag2)
IFEQ L00047
(L00040)
(line=16)
ILOAD 4 (d)
ISTORE 5 (sum)
(L00044)
(line=17)
GOTO L00054
(L00047)
(line=18)
FRAME-OP(0)
ILOAD 4 (d)
ILOAD 3 (c)
IADD
ISTORE 5 (sum)
(L00054)
(line=20)
FRAME-OP(0)
RETURN
```

図 1: Java プログラムのコード片とそれの命令表現

図 1 のソースコード上で 13 行から 18 行までの処理に関連する命令表現は，図 2 の示す赤

い枠範囲内になる．図 2 の命令表現をの命令間の静的な依存関係で示したのは図 3 になる．

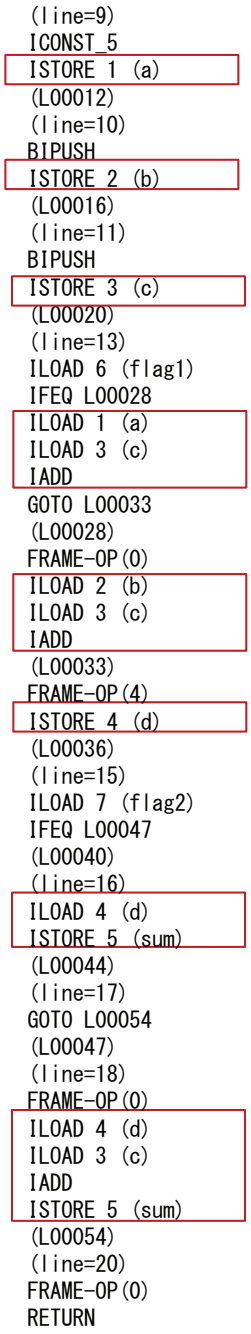


図 2: 図 1 のコード片の 13 行から 18 行までの処理に関連する命令群

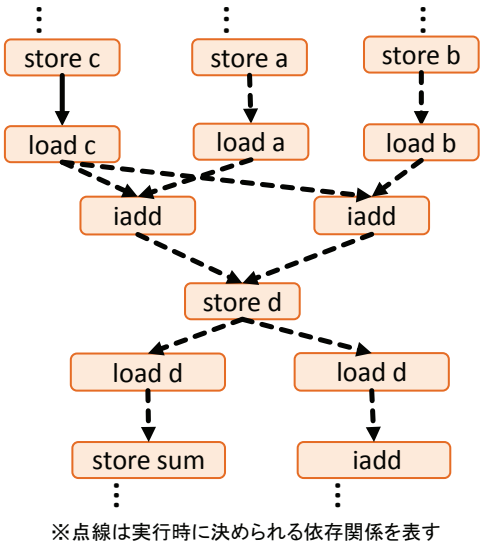


図 3: 図 2 の命令群に対する命令間の静的な依存関係

図 3 では store d 命令が実行時に iadd 命令から影響を受け，iadd 命令が load a 命令と load b 命令のどちらかと load c 命令から影響を受ける．実行時の状況によって store d 命令が 2 つの命令 load d のどちらかに影響を与える．プログラムの実際の実行では命令間に図 4 のような依存関係があったとする．

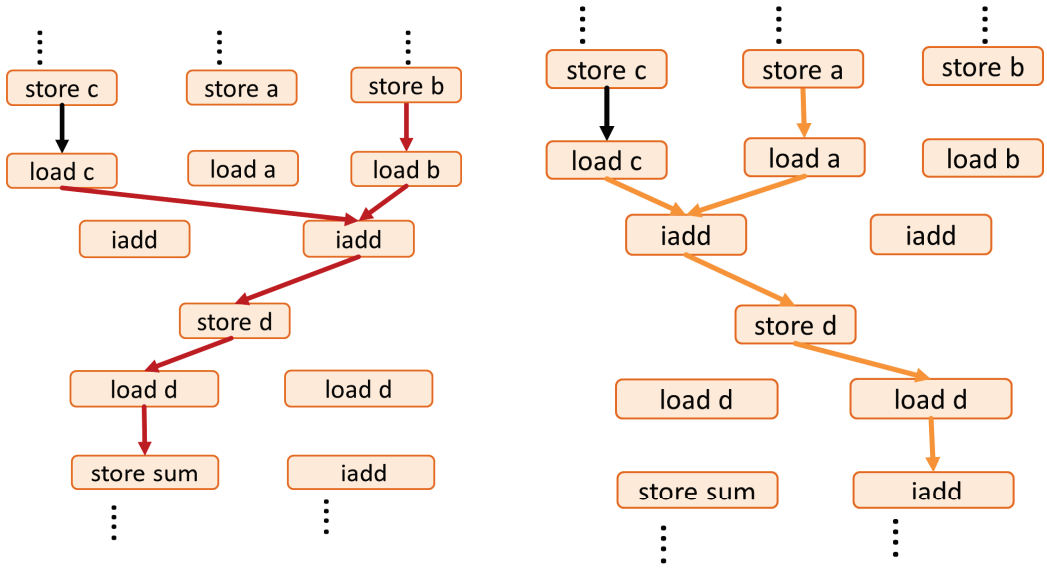


図 4: プログラムが実行された場合の命令間の依存関係（動的依存グラフからのコード片に関する部分の抽出）

図 4 では，load c,load a - iadd - store d - load d（右側）の順に進んだ場合と load c,load b - iadd - store d - load d（左側）の順に進んだ場合の 2 通りを示している．図 4 の実行トレースによる動的依存グラフを，命令間の依存関係単位で 3-gram 分解したとする．その場合，図 1 に示すコード片の 13 行から 18 行までの処理に関係する部分が図 5 のようになる．図 5 から分解後は load d 命令に load a と load b のどちらが影響を与えたかわからなくなる．また，図 4 のような実行があった場合，load c,load a - iadd - store d - load d（左側）経路も存在しなかったことを表現できない．

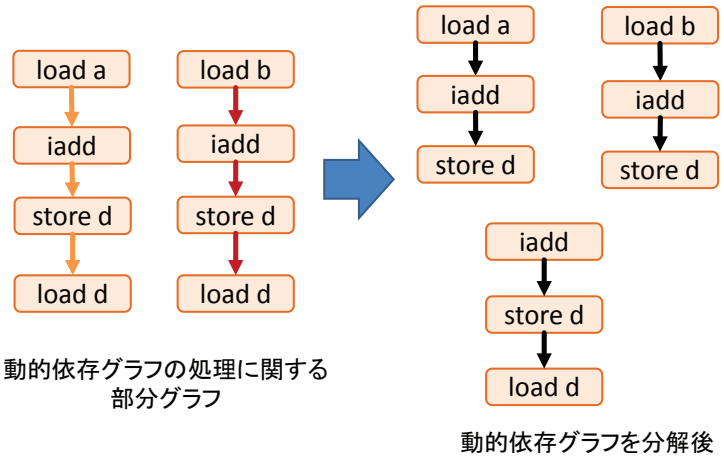


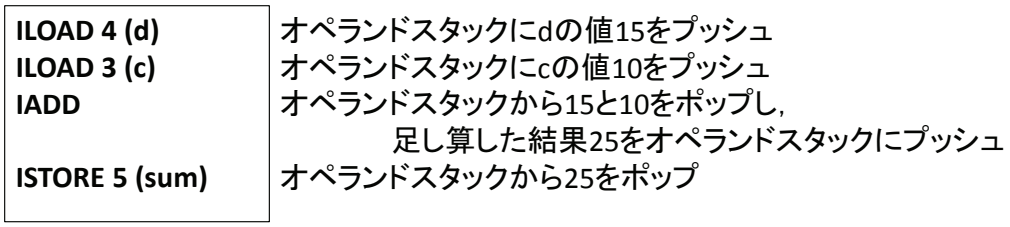
図 5: 処理に関する動的依存グラフの部分グラフとその分解

動的依存グラフの分解単位を長くすれば，詳細に動作表現ができるが，どれぐらい長くするかについて定まった値は存在しない．また，分解単位が長いほど，意味のある区切りで動的依存グラフを分解できているとは限らない．ソースコード上の構造によって，命令の実行が静的に決定されるものと動的に決定されるものがある．この静的に実行が決定される命令の連続したまとまりを 1 つのブロックとすると，プログラムの実行では，動的に実行が決定される命令によって，これらのブロックが連結されていく．つまり動的依存グラフをいくつかのブロック間の区間に区切ることができ，これらのブロックを互いに連結させたものと見なすことができる．動的依存グラフの分解単位をブロックにすると意味のある区切りで分解ができる．しかし，ブロックの長さはソースコード上の構造によって様々である．そのため，分解単位をある特定の長さにすることができない．そこで，本研究では，ブロックという概念を導入し，命令を頂点とする動的依存グラフをブロックを頂点とする動的依存グラフにする．ブロックの概要について次節で述べる．

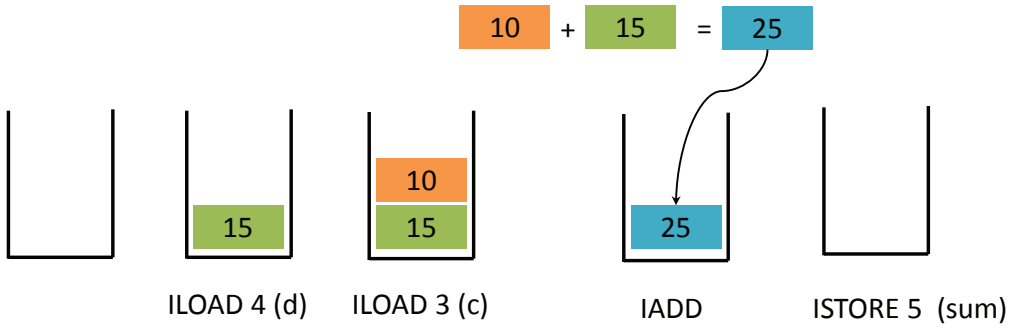
3.2.1 ブロックの定義

ブロックとは，ソースコード上で実行が静的に決められる命令の連続したまとまりである．同一メソッド内で次の 2 つの条件のいずれかを満たす場合ブロックとして区切りする．

- ある命令 p の実行によってオペランドスタックサイズが 0 になったときの，直前ブロックの終了地点から p までの（バイトコード上の）命令列
- メソッド呼び出し命令 q が実行されたとき，直前ブロックの終了地点から q までの



(a) sumがd+cから値を取得する処理のバイトコード表現とその説明



(b) sumがd+cから値を取得する処理の実行によるオペランドスタックの状態

図 6: 図 1 の 18 行で d=15 の場合の処理と sum=b+c の命令表現とその実行によるオペランドスタックの状態

（バイトコード上の）命令列（ただし，メソッド呼び出し命令とは `invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual` のことを指す.）

Java プログラムでは，メソッドの実行開始時点でオペランドスタックは空であるので，メソッドの動作はいくつかのブロック処理で構成されると見なすことができる．

以下に，ブロックの具体例と，ブロックで Java プログラムの動作を表現できることを説明する．Java プログラム実行において，代入文，条件文，メソッド呼び出しなどの（区切りがある）処理は複数の命令の実行で構成され，ブロックの代表的な例である．例えば，図 1 の 18 行の sum=d+c 代入文を例に説明する．d と c の値がそれぞれ 15 と 10 とする．その場合，sum=d+c 代入文の命令実行とそのときのオペランドスタックの状態を図 6 に示す．その動作は，まず，ILOAD 命令によってオペランドスタックに d，c の値がプッシュされ，IADD 命令によって b，c の値がポップされ，足し算された結果がプッシュされる．最後に足し算の結果を ISTORE 命令によってポップし，変数 sum に格納する．

メソッドの動作はいくつかのブロック処理で構成されるので，ゆえに，プログラム全体の実行がいくつかのブロック処理で構成されると見なすことができる．従って，動依存グラフをいくつかのブロック区間にわけ，各区間と区間の間を命令間の依存関係で分解することができる．

あるブロック A の N 回目の実行を A_N とするとブロック間の依存関係を次のように定めることできる．命令 i がブロック B に属し，命令 j がブロック C に属するとき，Dep(j_N,i_M) が成り立つ場合，B_N から C_M へブロック間依存関係が存在する．

3.2.2 ブロックを頂点とした 3-gram 表現

前節で述べたブロックを分解単位にすると意味のある区切りで動的依存グラフを分解できる．しかし，動的依存グラフの分解単位を 1 つのブロック処理にするとブロック処理間の依存関係がわからなくなるので，ブロック処理とブロック処理間の動作を表現できないそこで，分解単位を命令間の依存関係によって接続されたブロック区間の 3-gram にすると，あるブロック処理に影響を与えたものと，そのブロック処理が影響与えるものがわかるようになり，ある程度動作表現を保った分解ができる．そのため，実行時に実行トレースからブロック区間を結ぶ命令間の依存関係とそれらの命令の実行を実行動作を表現する情報として記録する.（つまり実行トレースからブロック処理を頂点とした，それらを結ぶ命令間の依存関係をエッジとしてグラフの情報が出力される）

3.2.3 動的依存グラフの分解の実装

提案手法は，比較する各実行トレースから記録された情報より得られたグラフ（頂点がブロックでエッジがそれを結ぶ依存関係を表す）を 3-gram に分解し，分解したものから重複するものを除去する．図 7 にプログラムの実行を表すブロック処理を頂点に持つグラフの例を示す．図 7 のグラフの頂点に記述しているのはブロックに含まれる命令列のうち，最後の命令が実行された時刻である．グラフの分解は次の 6 つのステップから成り立つ．以下に，その流れを説明する．

ステップ 1) グラフから頂点の時刻が一番大きい葉を選択

ステップ 2) その葉の親頂点をグラフから抽出

ステップ 3) 抽出した各親頂点の親をグラフからさらに抽出 or 葉が親を持たない場合ステップ 1）に戻る

ステップ 4) [祖先-親-葉] の依存関係を出力 or　葉の親が祖先持たない場合 [親-葉] 依存関係を出力

ステップ 5) 葉をグラフから消す

ステップ 6) グラフが頂点がある場合ステップ 1）に戻る

以降，図 7 を用いて各ステップの処理を説明する．

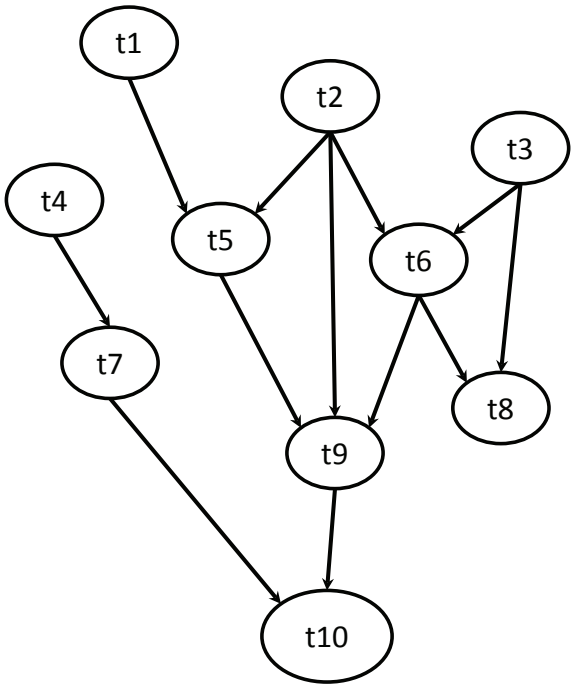


図 7: ブロック処理を頂点に持ち，プログラムの実行の全体を表すグラフの表現例

ステップ 1，ステップ 2，ステップ 3 の処理： ステップ 1 からステップ 3 までの処理の概要を図 8 に示す．

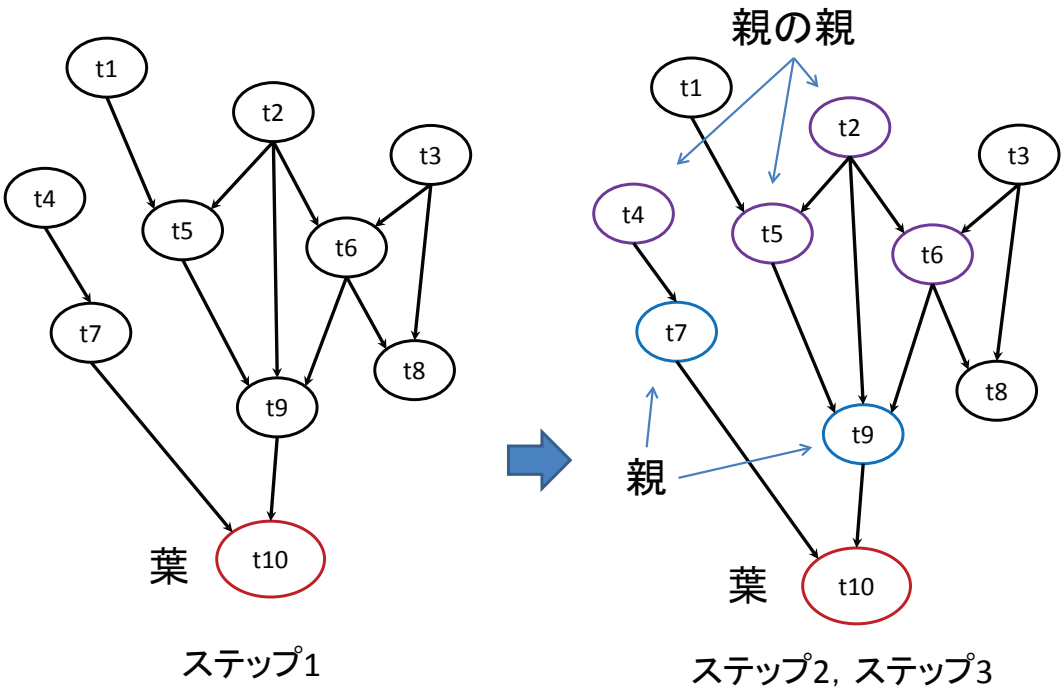


図 8: ステップ 1，ステップ 2，ステップ 3 の処理

ステップ 4 の処理：　ステップ 1 からステップ 3 までの処理の概要を図 9 に示す.

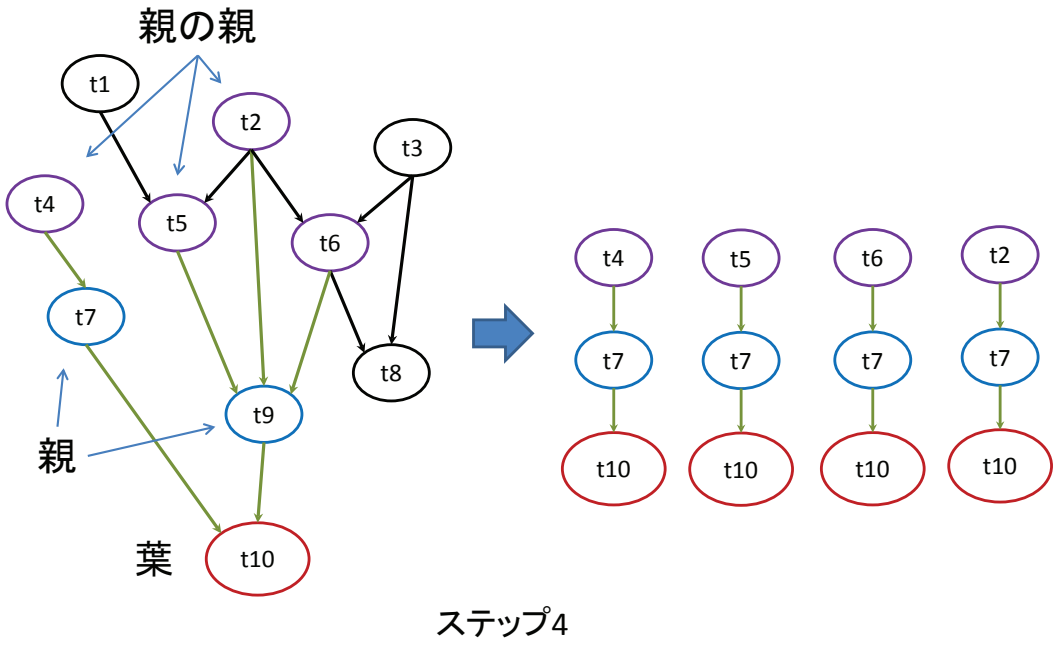


図 9: ステップ 4 の処理

ステップ 5，ステップ 6 の処理：　ステップ 1 からステップ 3 までの処理の概要を図 10 に示す.

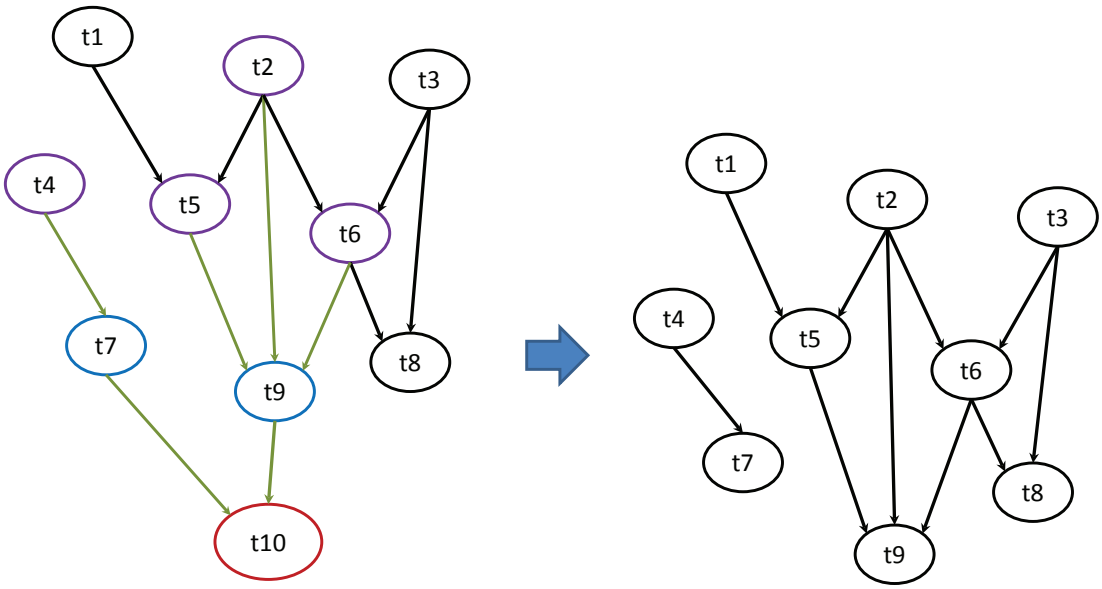
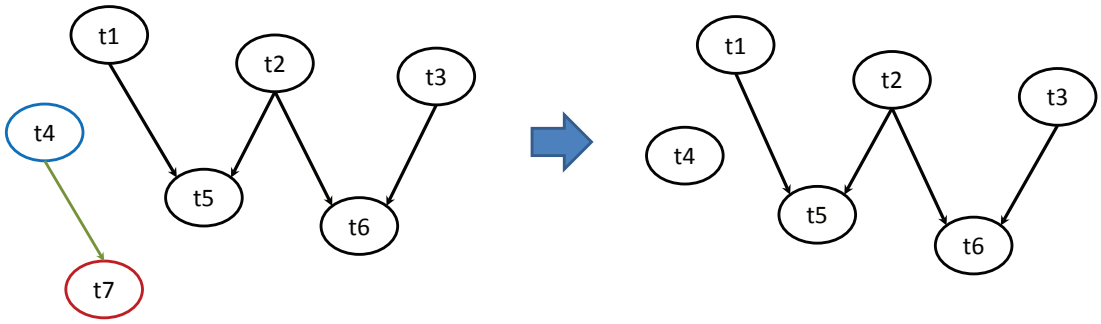
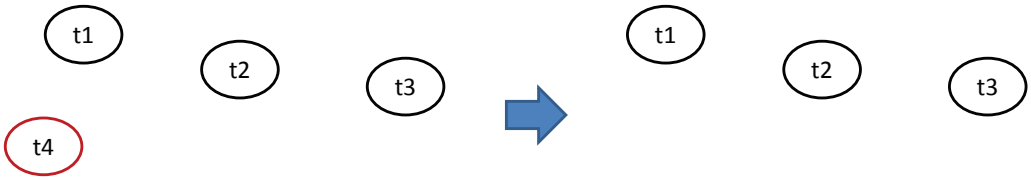


図 10: ステップ 5 の処理

そして，ステップ 3 で葉が親を持たない場合，ステップ 4 で親が祖先持たない場合の処理を図 11 に示す．図 11 の (a) では，頂点 t7 の説明のため，頂点 t8，t9 の処理をスキップしている．図 11 の (a) では，t4-t7 の依存関係を出力する．図 11 の (b) では，頂点 t4 の説明のため，頂点 t5，t6 の処理をスキップしている．図 11 の (a) では，出力することなく，次のステップへ進む．



(a)時刻が一番大きい葉の親が祖先を持たない場合



(b)時刻が一番大きい葉が親を持たない場合

図 11: ステップ 3 で葉が親を持たない場合，ステップ 4 で親が祖先持たない場合の処理

3.3 分解した動的依存グラフの比較

グラフ（頂点がブロック処理でエッジがそれらを結ぶ命令間の依存関係を表す）の分解が終わったら，分解単位で重複するものを除去し，分解したグラフ同士を比較する．分解単位は命令間の依存関係で接続された 3 つのブロック区間の並びであり，分解したグラフはその集合である．

分解単位を集合にする作業：

分解単位に含まれるあるブロック B の i 回目の実行 B_i をブロック B の最後の命令の命令番号に置き換える．つまり，分解単位は $B_L -> C_M -> D_N$ で，B，C，M のっ最後の命令はそれぞれ j，k，l あれば，$j -> k -> l$ に変換する．すべての分解単位の変換が終わったら，分解したグラフを分解単位の集合にする．

比較する作業：

比較する 2 つの実行トレースを仮に実行トレース 1 と実行トレース 2 とすると，比較作業とは，実行トレース 1 と実行トレース 2 に共通するものと各実行トレースにしかないものに分類することである．実行トレース 1 と実行トレース 2 に対する分解したグラフが仮に図 12

の (a) のようになったとすると，その分類結果は図 12 の (b) のようになる．

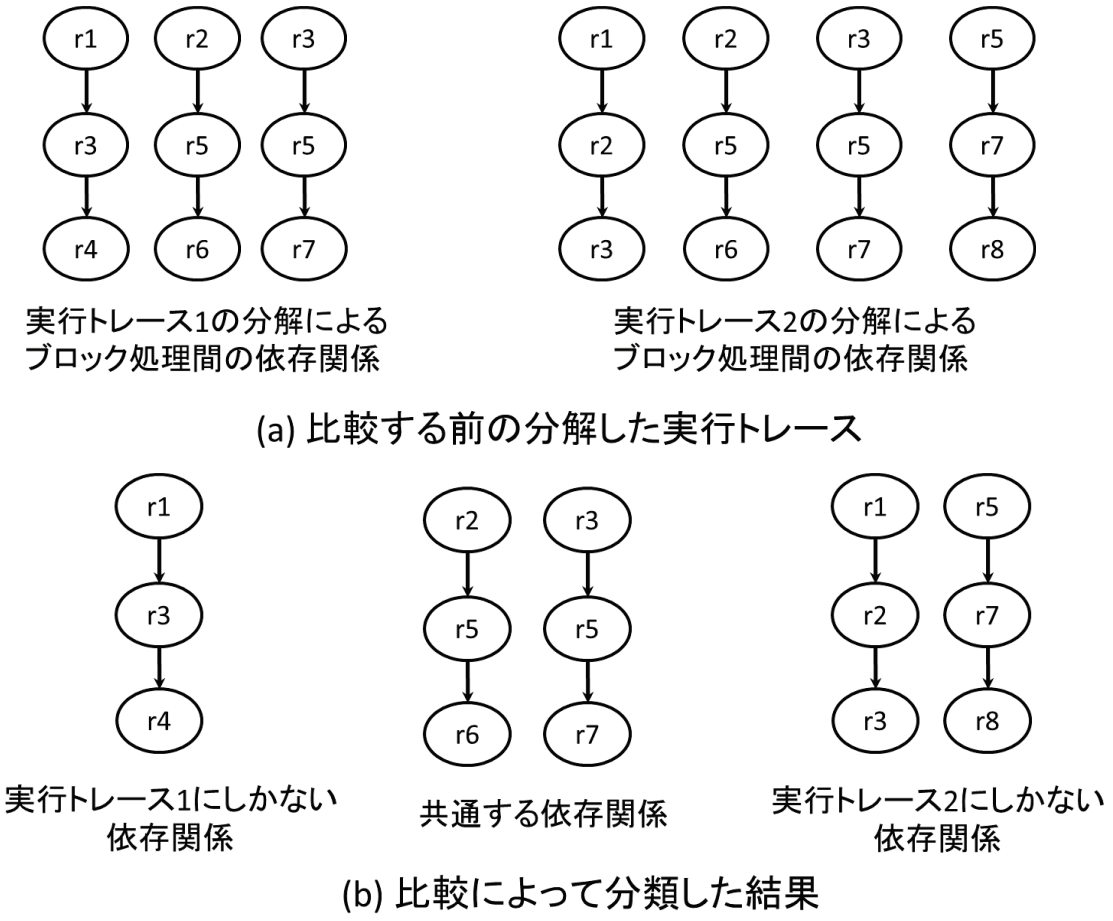


図 12: 分解した 2 つの実行トレースの比較

最後に，分類した結果の各頂点の命令番号を実行された命令の実行が持つ情報（クラス名，メソッド名，行番号，バイトコード命令）に変換する．

提案手法による比較によって,「同一」と判定された場合は 2 つ実行トレースは以下の項目に関して等しいと言える．

- 行カバレッジ
- ブランチカバレッジ

- ループの 1 回目の実行，2 回目以上の実行の対応
- def-use カバレッジ

3.4 提案手法の実装

本節では，提案手法の実装を説明する．提案手法の実装は「実行トレースから実行動作を表現する情報の取得」とその解析から構成される．以降，それぞれの実装を詳細に説明する．

3.4.1 ブロックを頂点に持つグラフの取得

ブロックを頂点に持つグラフの取得する処理の概要を図 13 に示す．この処理の実装では selogger，replayer という 2 つの既存ツールを使用した．selogger はプログラムの実行再生に必要な情報をイベント単位でプログラム実行中に記録し，それを実行履歴としてログファイルを出力するツールである．イベント単位は，メソッドの開始，メソッドの終了，フィールド代入など粗い単位である．replayer は selogger の出力したログファイルとを元にプログラムの実行を再現するツールである．replayer は対象となるプログラムの実行ファイルからバイトコード命令を取得し，selogger のログファイルから実行再生に必要な情報を抽出して，プログラムの実行を再現する．

図 13 の流れを説明する．まず，selogger で対象プログラムの実行ファイルにログとるためのコードを追加し，それを出力する．次にログをとるためのコードが追加された実行ファイルを実行させ，動作に関するログファイルを取得する．そのログファイルと実行ファイルを用いて，replayer で実行を再生する．提案手法の実装では，実行を表す情報を取得するために，replayer のコードに変更を加え，実行を再生しながら，ブロック処理とそれらの間の依存関係（実行を表す情報）を出力するようにしている．

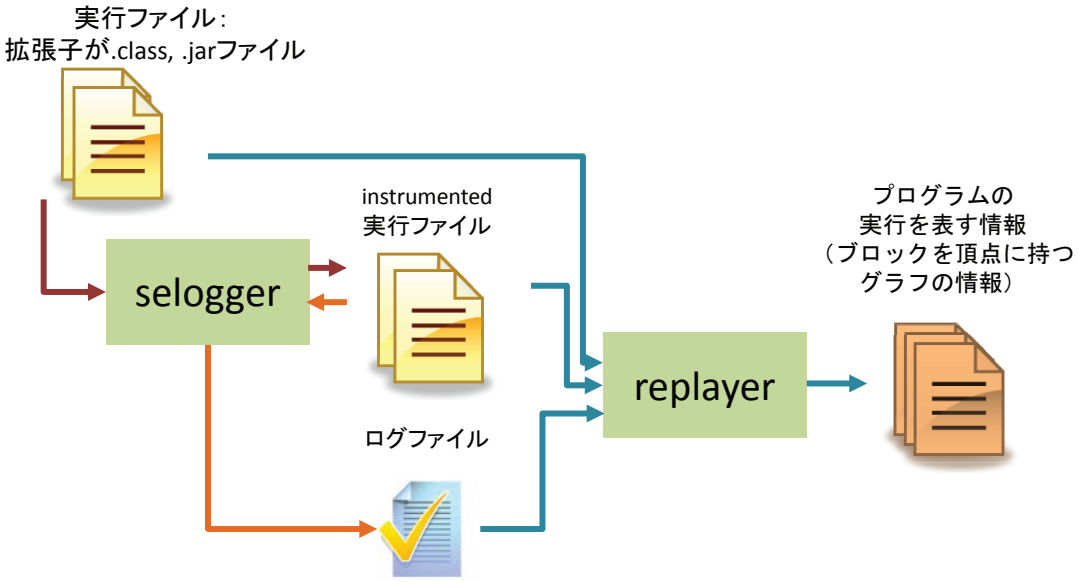


図 13: 実行トレース取得の概要

3.4.2 ブロックを頂点に持つグラフの解析

実行トレースの解析の流れを図 14 に示す．ブロックを頂点に持つグラフの解析では，まず，前のステップで出力したファイル（図 13）を読み込みブロック処理間の依存関係を取得しながら，ブロック区間の 3-gram 依存関係で分解していく．分解が終わったら，ブロック区間の 3-gram 依存関係の頂点を（ブロックに含まれる最後の）命令に置き換えてから，集合にし，重複するものを取り除く．その後,「集合にした 3-gram 依存関係の情報」を含むファイル同士の内容を比較し，その結果を，3 つのファイル（共通する依存関係を含むファイル，トレース 1 にしかない依存関係を含むファイル，トレース 2 にしかない依存関係を含むファイル）に出力する．

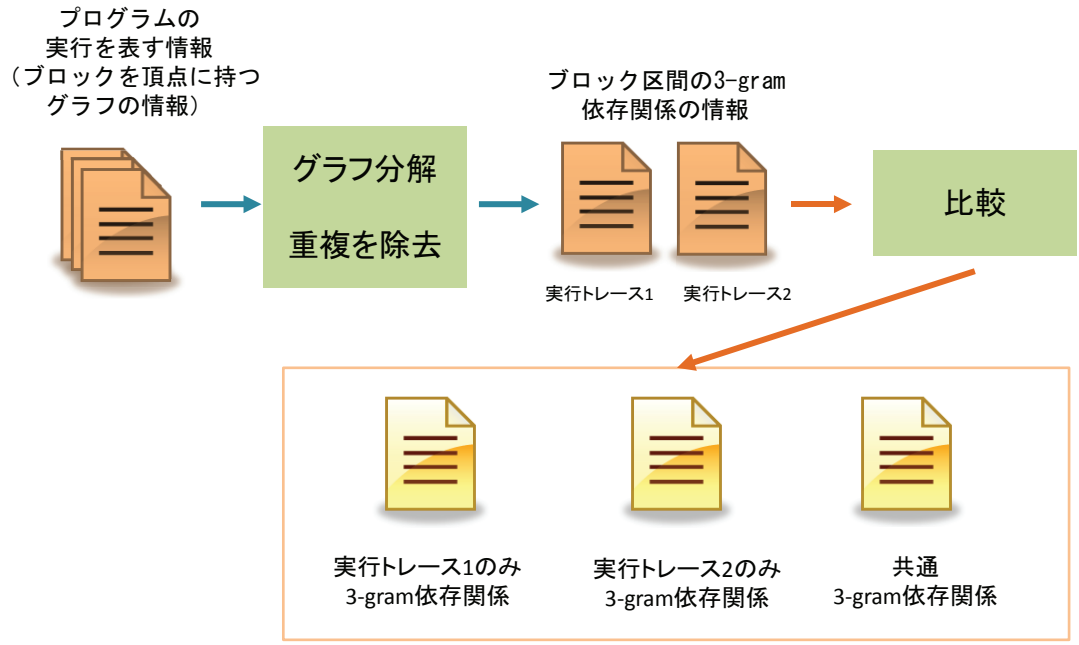


図 14: 実行トレースの解析の流れ

4　ケーススタディ

本研究では，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 [3] 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 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

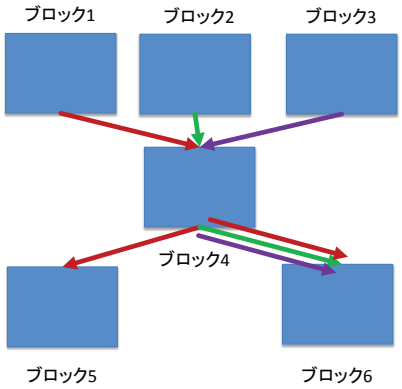


図 15: ブロックの 2-gram 依存関係で動的依存グラフを分解すると検出することができない実行例

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．

表 2 には，3-gram 集合を比較し，各実行トレース固有の 3-gram と共通 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が異なる入力に対して，異なる動作をしているのがわかる．特に batik においては，default 入力は small 入力と同じ動作を 3 回繰り返すだけであると我々は予測していたが，実際には default 側には多くの追加処理があり，また small にしか含まれないような処理も存在することが各印できた．

本研究では 3-gram の集合を比較対象に使用したが，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も実行トレースには含まれていた．あるブロック B の実行に影響を与えるブロックが c 個，ブロック B の実行が影響を与えるブロックが d 個あるとすると，その実行経路を表現した 3-gram は最大で $c \times d$ 個存在するが，2-gram では隣

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	3-gram ファイル	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

表 2: 各実行トレースにおける 3-gram の比較結果

アプリケーション名	入力オプション	実行トレース固有の 3-gram の数	共通 3-gram の数
batik	small	405	14413
	default	7,895	
fop	small	1,256	10676
	default	4,959	

表 3: ブロックに影響を与えたものの個数とそのブロック影響を与えたものの個数の全組み合わせ数が実際に実行トレース上に現れた組み合わせ数より多いケース

アプリケーション名	入力オプション	全組み合わせ数が実際組み合わせ数より多いケース（個数）
batik	small	191
batik	default	296
fop	small	161
fop	default	171

接したブロック間の $c+d$ 通りの関係までしか取り扱うことができない．例えば，図 15 のような 3-gram 集合が存在したとする．6 つの実行ブロックに対して，$1 \to 4 \to 5$, $1 \to 4 \to 6$, $2 \to 4 \to 6$, $3 \to 4 \to 6$ という 4 つの実行経路が観測されている．このうち，$1 \to 4 \to 6$ という実行経路は，2-gram 集合の場合では存在しても，存在していなくても変化しないため，集合の差分からこのような経路の存在を知ることはできない．このように，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の存在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 15 におけるブロック 4 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．このような条件を満たした 3-gram の実例を図 16 に示す．図 16 では，青い四角形で頂点を表しており，[クラス名#行番号] の形で命令ブロックに対応するソースコードの位置を表現している．また，頂点の外に，その実際のコード片を示した．各頂点をつなぐ辺は 3 色使用しているが，それぞれの色が異なる実行経路を表している．

最後に表 2 に示した結果から 1 つの事例を図 17 に紹介する．図 17 では，batik の small 入力と default 入力における，動作の差を示している．default 入力の実行では，679 行目の

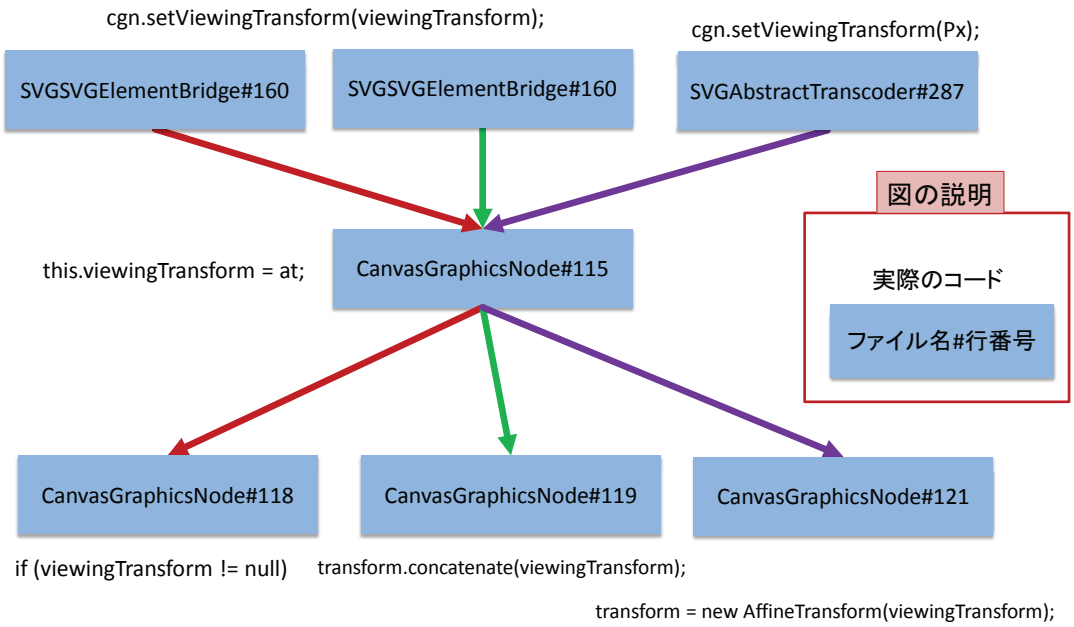


図 16: 2-gram 依存関係で検出できないかつ動作が異なる 3-gram 依存関係の例

if 文の条件節は 1 つ目の条件文で false になり，実行が 684 行へ移動している．small 入力の実行では，679 行目の if 文の条件節は 3 つ目の条件文まで実行が進み，その結果が false になって実行が 684 行へ移動している．図 17 の事例より， 3-gram 依存関係で動作のわずかな違いを抽出できることを確認できる．

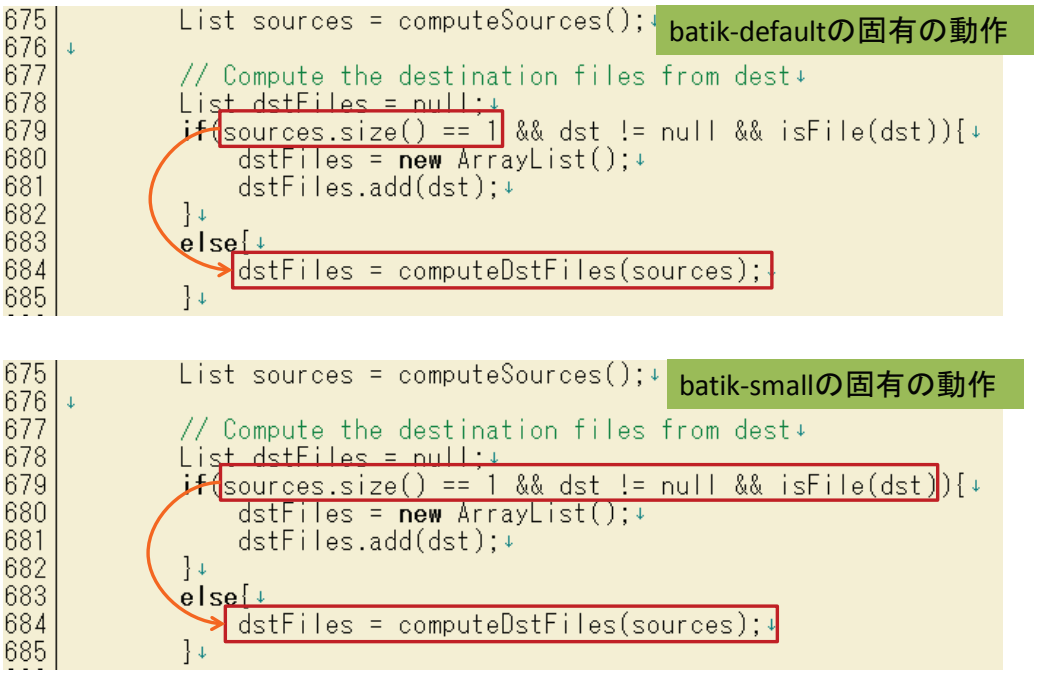


図 17: 異なる入力に対する動作の違いの事例

5　まとめと今後の課題

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 の集合に分解して比較を行っている．ソフトウェアテストにおけるカバレッジ計算の概念が 1-gram あるいは 2-gram に相当しており，提案手法はそれよりも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．ケーススタディでは，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た．結果として，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，各アプリケーションのテストケースによる差異を抽出できることを確認した．また，動的依存グラフの 2-gram では表現できない動作が実際に数多く実行トレースに含まれていることも確認した．

今後の課題として，動的依存グラフの分解単位を 4-gram，5-gram などに伸ばした場合の効果の調査が挙げられる．また，本研究ではオペランドスタックを使用してデータを受け渡す連続したバイトコード命令のブロックを頂点としたが，このブロックの範囲の選び方を変更した場合に得られる効果を調査することも考えられる．また，提案手法をより多くの種類のアプリケーションに対して適用し，その有効性を調査することが挙げられる．

謝辞

本研究において，常に適切な御指導及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．

本研究において，随時適切な御指導及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松下誠准教授に心より深く感謝いたします．

本研究において，終始適切な御指導及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心より深く感謝いたします．

最後に，その他様々な御指導，御助言等を頂い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, 2009.

[2] 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, 1990.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, 2006.

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

[5] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, Vol. 25, No. 1, pp. 53–95, 2013.

[6] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, Vol. 29, No. 3, pp. 210–224, 2003.

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

[8] A.J. Ko and B.A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, pp. 301–310, 2008.

[9] Bil Lewis. Debugging backwards in time. In *Proceedings of the 5th International Workshop on Automated Debugging*, 2003.

[10] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficientvisualization of execution paths and states in java. In *Proceedings of the 22nd International Conference on Program Comprehension*, 2014.

[11] Andriy V Miranskyy, Nazim H Madhavji, Mechelle S Gittens, Matthew Davison, Mark Wilding, and David Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *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.

[12] Jonas Trumper, Jurgen Dollner, and Alexandru Telea. Multiscale visual comparison of execution traces. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pp. 53–62, 2013.

[13] Andy Zaidman. *Scalability solutions for program comprehension through dynamic analysis*. PhD thesis, University of Antwerp, 2006.

[14] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging, the Second Edition*. Morgan Kaufmann, 2011.

[15] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 81–91, 2006.

[16] Hong Zhu, Patrick a. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, Vol. 29, No. 4, pp. 366–427, December 1997.

[17] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラム実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, Vol. 24, No. 3, pp. 153–169, 2007.