プログラムの変更前後での実行履歴の差分検出手法

伊藤芳朗[†1]　渡邊結[†1]
石尾隆[†1]　井上克郎[†1]

開発者は，デバッグ作業などにおいてソースコードを変更したとき，プログラムの振舞いの変化が意図したものであるかを確認する必要がある．そのために，ソースコードを変更した前後で同じ操作手順を実行し，振舞いの差異を調査することになるが，外部から観測が困難な変更については，実行中に生じる膨大な量のメソッド呼び出しの系列を比較するなどの作業が必要である．そこで本研究では，ソースコードを変更した前後で同じ操作手順を実行して得られた 2 つの実行履歴に対して，それらの間の差分を木構造の比較により検出，可視化する手法を提案する．画像編集ソフト JHotDraw を対象にケーススタディを実施した結果，可視化された差分と，バージョン間の変更点を対応づけられることを確認した．

Detecting Difference between the Execution Trace Extracted from Two Versions of a Program

Yoshiro Ito,[†1] Yui Watanabe,[†1] Takashi Ishio[†1]
and Katsuro Inoue [†1]

It is necessary to confirm whether the change of behavior of a program is intended when developers modify souce code for debugging. To check the difference of the behavior of the program, they execute the modified program under the same operating condition. However, it is necessary to compare a huge amount of method call events in execution traces when they can not directly observe their behavior. This paper proposes a method to detect and visualize the difference by comparing tree structures between the two execution traces that are obtained by executing the two versions of a program under the same operating condition. We conducted a case study for a drawing editor JHotDraw. As a result, we confirm that the visualized difference of the execution traces corresponding to the source code difference.

1. まえがき

開発者は，デバッグ作業などにおいてソースコードを変更したとき，プログラムの振舞いの変化が意図したものであるかを確認する必要がある．そのために，ソースコードを変更した前後で同じ操作手順を実行し，振舞いの差異を調査することになるが，欠陥の修正によってデータの値などが変化した影響は，様々なデータを経由してプログラムの出力などに間接的に影響を与える[3] ため，直接の観測が難しい 場合がある．一方で，オブジェクト指向プログラムの振舞いを理解するためには，プログラムの実行を観測し，実際の振舞いを可視化することが有効であることが知られている[19]．実行時の振舞いの変化を可視化することでソースコードの変更による直接的な影響を確認可能するために，実行履歴を記録し，その変化を可視化することが有効であると考えられる．

本研究では，ソースコードを変更した前後で同じ操作手順を実行して実行履歴を取得し，その実行履歴の差分を検出する手法を提案する．検出した実行履歴の差分を可視化することで，プログラムの振舞いの変化が意図したものと一致しているかを判断することが可能になる．具体的には，取得した 2 つの実行履歴中のメソッド呼び出し関係をそれぞれ順序木によって表現し，2 つの木の比較を行うことで実行履歴の差分を検出する．実行履歴を可視化する際には，利用者がわかりやすいように実行履歴の差分を強調表示する．また，実行履歴の差分を抽出することでプログラムの振舞いの変化を発見しやすくする．本手法の対象とするソースコードの変更の場面は主にデバッグ作業を想定している．デバッグ作業によるソースコードの変更では，差分は小さくなると考えられるため，木構造の差分だけを検出し，可視化する．

我々の研究グループでは，java プログラムの実行履歴からシーケンス図を生成するシステム Amida[20] を開発している．本研究ではこの Amida を改造することで提案手法を実装した．そして，実際に画像編集ソフト JHotDraw[9] の 2 つのバージョンで同じシナリオを実行した実行履歴を取得し，提案手法を適用した結果，表示された差分から 2 つのバージョン間での変更の影響を特定できることを確認した．

以降，2 節ではこの研究の背景を述べ，3 節で提案手法の説明をする．4 節では行ったケーススタディとその結果について述べ，5 節では関連研究を紹介する．最後に 6 節で本研究の

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

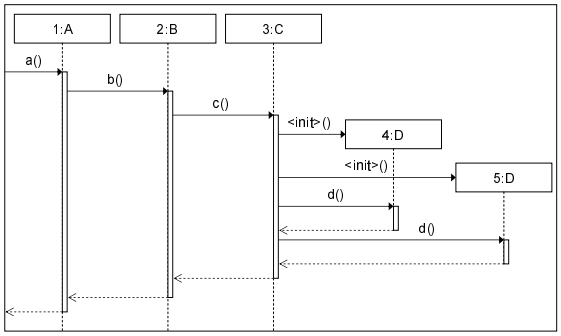

図 1 シーケンス図

まとめと今後の課題を述べる．

2. 背　　景

2.1 オブジェクト指向プログラム

オブジェクト指向プログラムでは，クラスを用いて処理の抽象化を行い，また継承や多態性などを利用して，オブジェクト間のメソッド呼び出しによってソフトウェアの機能を実現する．このようなシステムの振舞いを理解するためには，ソースコードのような静的な情報を解析する手法と，実行履歴のような動的な情報を解析する手法がある．ソースコードを解析する手法では動的束縛などの動作を完全には知ることができない場合があるため，実際にソフトウェアを実行し，どのオブジェクトが呼び出されたかを記録した実行履歴を解析する手法が有効であり[19]，実行履歴をシーケンス図などの形式で可視化する手法が広く研究されている[4),18),20)]．

シーケンス図は Unified Modeling Language(UML)[17] で定義されているインタラクション図の 1 つで，オブジェクト間のメソッド呼び出しやオブジェクトの生成などのメッセージ通信を，時系列に沿って示した図である．この図を作成することで，設計者はシステムが実際に動作する際のオブジェクト間の関連を，他の設計者や開発者に示すことが可能である．プログラムの振舞いの可視化の研究においても，その動作を示すために利用されている．

2.2 Amida

我々の研究グループは，オブジェクト指向プログラムにおけるオブジェクト実行時の振舞いを視覚的に表現し，プログラムの理解支援を行うために，プログラムの実行時に観測した実行履歴を基にシーケンス図の生成を行うツール Amida を開発している[20)]．Amida は Java プログラムのメソッド呼び出しを実行履歴として取得するプロファイラと，その実行履歴を解析してシーケンス図を生成，表示するビューアからなるツールである．図 2 は Amida のスクリーンショットで，図の左側にシーケンス図が表示されている．ただし，実行履歴から生成したシーケンス図は巨大なサイズになってしまうため，上部にオブジェクトを表示し，その下にシーケンス図の一部分を表示する．図の右側はシーケンス図の縮小図を表している．また，Amida にはシーケンス図上の繰り返し処理や再起呼び出しなどのループの部分を圧縮して表示する機能が実装されている．

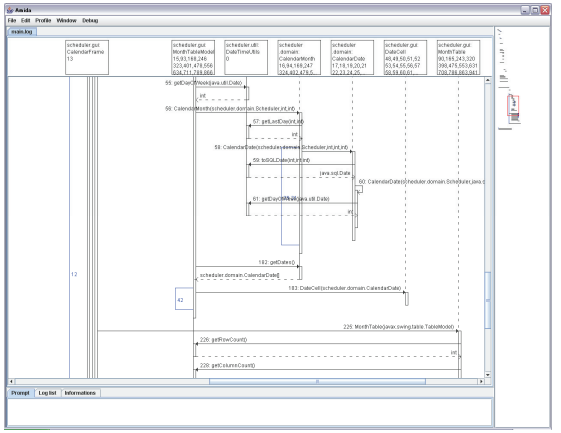

図 2 シーケンス図生成システム Amida

2.3 実行履歴の比較

プログラムの実行履歴は実行時の動作を記録したものであるため，利用者が同じシナリオで実行しても実行履歴が変化する場合がある．その要因はソースコードの変更，実行環境の変化，入力データの変化の 3 つに分類できる．

ソースコードが変更された場合，変更に応じてプログラムの振舞いが変わるため，実行履歴が変化する．例えば，メソッド内にメソッド呼び出しを 1 つ追加すると，追加されたメソッド呼び出しと，そのメソッド内で呼び出される他のメソッド呼び出しが実行履歴に追加される．また，メソッドのオーバーライド関係の変更を行うと，呼び出し側のコードが変更されていなくとも，動的束縛によって動作が変わることがある．

同じ実行をした 2 つの実行履歴を比較した場合，実行履歴の差分が検出されたときに，そこになんらかの変化があり，その変化に注目することで，ソースコードの変化や入力データの変化などのプログラムの振舞いの変化の要因を発見できると考えられる．

本研究では，実行履歴の変化の要因のうち，デバッグ作業におけるソースコードの変更によるものを取り扱う．欠陥の修正によるメソッド呼び出しやデータの値などの変化は様々なデータを経由してプログラムの出力などに間接的に影響を与える[3] ため，ソースコード編集による直接の影響を確認しておくことが，修正の妥当性を確認するために重要となる．

プログラムの開始から終了までのすべてのメソッド呼び出しを記録すると，膨大な量になることが知られている[12),14),15)]．しかし，バグ修正におけるソースコードの変更は，多くの場合 1 つあるいは 2 つのファイルに対するものであり[7)]，実行履歴の変化は，実行履歴全体の中でもごく一部にのみ現れるものであると推測される．そのため，実行履歴の差分を自動的に検出，可視化することにより，ソースコード編集による影響を，開発者が効果的に確認することが可能であると期待される．

3. 提 案 手 法

本研究では，2 つの実行履歴を比較し，実行履歴の差分の位置を特定する手法を提案する．特定した実行履歴の差分を可視化することで，ソースコードの変更によるプログラムの振舞いの変化が意図したものと一致しているかを判断することが可能になる．具体的には，取得した実行履歴からオブジェクト間のメソッド呼び出し関係を取得し，順序木に変換する．そして，ソースコードの変更前後の順序木の実行履歴の比較を行い，ノード間の対応関係を求めることで実行履歴の差分を検出する．

検出した実行履歴の差分をシーケンス図を用いて可視化することで利用者に分かりやすい形で提供する．ただし，可視化する際に実行履歴の差分だけを可視化しても，その部分がどのような状況で呼び出されたのかが判断できないため，実行履歴全体をシーケンス図として生成し，実行履歴の差分を強調して表示する．また，実行履歴が膨大になることを考慮し，可視化した実行履歴の中から振舞いが変化した部分のみを抽出した可視化も行う．

提案手法は，Java プログラムの実行履歴を可視化するシステムである Amida の存在を念頭に置いて，Java プログラムの実行履歴に対応した手法となっているが，同様のメソッド呼び出し情報が取得可能な実行環境を持つプログラミング言語などにも適用可能である．

3.1 実行履歴の取得

提案手法が対象とする実行履歴は，実行されたメソッド呼び出しイベントの系列である．対象とするプログラムの実行を観測し，オブジェクト間のメソッド呼び出しに関する情報を記録する．本手法では，Amida のプロファイラを使用して実行履歴を取得する．Amida では，個々のメソッド呼び出しについて，メソッド開始時にスレッド番号，タイムスタンプ，クラス名，オブジェクト ID，メソッド名，引数の型，返り値の型を呼び出された順に記録する．コンストラクタの呼び出しは，<init>という名前のメソッド呼び出しとして表現している．また，メソッド終了時には終了記号を記録する．引数の型を記録するのは，メソッドがオーバーロードされて同名のメソッドが複数存在する場合に，メソッドを特定するためであり，実行時に引数として与えられた値については記録しない．これらの情報を用いることで，実行時に呼び出されたオブジェクトとメソッドを特定し，メソッド間の呼び出し関係を木構造に変換することが可能となる．

3.2 メソッド呼び出し関係木の構築

最初に，取得した実行履歴を，メソッドの呼び出し関係を表現した順序木に変換する．実行履歴は，呼び出された全てのメソッドの開始と終了を記録しているため，あるメソッド M_1 が別のメソッド M_2 を呼び出す場合，実行履歴の中では，メソッド M_1 の開始，メソッド M_2 の開始，メソッド M_2 の終了，メソッド M_1 の終了の順に記録される．そのため，メソッドの開始と終了の順序から全てのメソッドの呼び出し関係が分かる．このメソッド呼び出し間の関係を木構造にし，これをメソッド呼び出し関係木と呼ぶ．

メソッド呼び出し関係木のノードは実行履歴に記録されている 1 つのメソッド呼び出しイベントとする．実行履歴中で，あるメソッド M_1 が直接メソッド M_2 を呼び出している場合，メソッド M_1 の呼び出しを親，メソッド M_2 の呼び出しを子とする．また，あるメソッド M_1 がメソッド M_2 とメソッド M_3 を M_2 ， M_3 の順で呼ぶとする．このとき，メソッド M_2 と M_3 の呼び出される順番を，そのままノードの順序として保存する．

3.3 実行履歴の差分の検出

次に，2 つの実行履歴から得られた 2 つのメソッド呼び出し関係木の比較を行う．ソースコードの変更や実行環境の変化，入力データなどの要因により実行履歴は変化するが，デバッグ作業におけるソースコードの変更によって生じる変化は小さいと仮定し，メソッド呼び出し関係木の比較には，トップダウン方式を採用し，木の根から順番に処理を進めていく．

メソッド呼び出し関係木の比較では，一方のメソッド呼び出し関係木から部分木を取り出し，もう一方のメソッド呼び出し関係木からその部分木と一致する部分を探索する．部分木 T_a と T_b があり，それぞれの根となるノードを r_a，r_b とするとき，2 つの部分木が等しい条件は以下のように設定する．

- ノード r_a と r_b が等しい．
- r_a と r_b の子ノードの数が等しい．
- r_a の i 番目の子ノードを根とする部分木と r_b の i 番目の子ノードを根とする部分木が等しい．

ここで，ノード r_a と r_b が等しいとは，次の 2 つの条件を両方とも満たす場合である．

- ノード r_a と r_b のクラス名，メソッド名，引数の型名の列，戻り値の型名が等しい．
- ノード r_a と r_b の呼び出し元クラス名（r_a と r_b の親ノードのクラス名）が等しい．

なお，クラス名や型の一致は，完全限定名の比較により判定する．

ノードの一致条件には，オブジェクト ID を使用しない．オブジェクト ID はメモリアドレスと生成順序を元に付与されるので，複数のスレッドが同時に実行される状況では同じ役割のオブジェクトでも実行ごとにオブジェクト ID が変わってしまう可能性があるためである．オブジェクト ID を比較しても，2 つのメソッド呼び出し関係木から同じ動作をするオブジェクトの対応をとることができない．

部分木の一致の条件から，等しい部分木の探索にはまず根となるノードと等しいノードを探索する．図 3 は 2 つの部分木が等しいかを探索する例である．図中のノードの番号が等しいとき，ノードの一致条件を満たしているとする．図の T_1 のうち四角形で囲まれた部分木が，T_2 の中に一致する部分木があるかを探索する．まず，根であるノード 1 と一致するノードを探索する．この探索はトップダウンに行うため，まず部分木 T_2 の根であるノード 0 と等しいか判定を行う．しかし，この 2 つのノードは一致しないため，ノード 0 の子ノードに移る．次のノードは共に 1 であるため，根となるノードと等しいノードを発見することができた．ここで，発見したノードが根となる部分木が探索したい部分木と一致するかを判定する．根となる 2 つのノードの子ノードの数を調べると共に等しいため，子ノードを根とする部分木がそれぞれ等しいかの判定を行う．同様に，ノード 2 を根とする部分木とノード 3 を根とする部分木が一致するかを判定する．図 3 では，ノード 2 を根とする部分木とノード 3 を根とする部分木は共に等しいため，部分木 T_1 と T_2 に含まれるノード 1 を根とする部分木は等しいと分かる．一致する部分木が見つかった場合，2 つの部分木に含まれる全てのノードに一致する部分があることをマークしておく．

この後，他の全ての部分木でも同様の探索を行う．全ての部分木で探索が終わったときに，一致する部分木があれば，その部分木に含まれるノードはマークされている．そのため，マークされていないノードは一致する部分がない，すなわち，実行履歴の変化部分であると分かる．

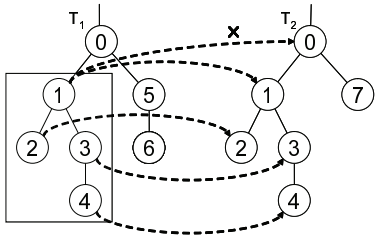

図 3 メソッド呼び出し関係木の比較

3.4 実行履歴の差分の可視化

実行履歴の変化を検出したときに，変化部分をそのまま利用者に提示しても，オブジェクト間のメッセージ通信や，他のオブジェクトとの関連性を理解することは難しい．そこで，実行履歴をシーケンス図として可視化し，変化となる部分を強調して表示する．

シーケンス図の生成手法は Amida[20] を参考にし，3.3 節で求めたマークの有無で強調表示を行うかどうかを判定する．本手法では，メソッド呼び出し関係木のノードは 1 つのメソッド呼び出しであり，ノード単位で変化部分であるかどうかの検出を行う．そのため，強調表示をするノードをシーケンス図にする際に，そのノードのオブジェクトへのメソッド呼び出しと呼び出されたオブジェクトの実行区間を強調して表示する．また，強調表示はシーケンス図上で表示する色を変えることで行う．

図 4 は図 3 にメソッド呼び出し関係木で示した 2 つの実行履歴をそれぞれシーケンス図として可視化し，実行履歴上の変化を強調表示したものである．図 4 では，ノード i をオブジェクト i へのメソッド呼び出しとして記述している．図 3 のノード 1 を根とする部分木は一致するとマークされるため，オブジェクト 0 からオブジェクト 1 への呼び出しとオブジェクト 1 からオブジェクト 2，3 への呼び出し，オブジェクト 3 からオブジェクト 4 への呼び出しは通常の表現で記述する．しかし，メソッド呼び出し関係木のそれ以外のノードは一致する部分がないため，図 4 のように赤色で強調表示を行う．

3.5 実行履歴の変化部分の探索と抽出表示

可視化したシーケンス図のサイズが巨大であれば，差分も膨大な数になる可能性がある．そこで，可視化したシーケンス図から差分を取り出して表示を行う．このとき，メソッド呼び出し関係木内のノードを全て調べることで実行履歴の差分を探索する．ただし，該当するメソッド呼び出しだけを表示すると，呼び出し元となるオブジェクトやその前後のメソッド呼び出しが分からないため，実用性に欠ける．そのため，差分として検出されて部分木の親

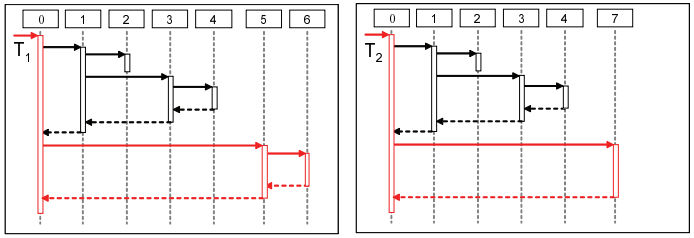

図 4 シーケンス図の比較

ノードを根とする部分木の単位で，シーケンス図による可視化を行うものとした．このときに，親となるオブジェクトへのメソッド呼び出しが同一の場合，同一のシーケンス図が複数表示されないように重複したメソッド呼び出しを表示しないようにする．

3.6 マルチスレッドへの対応方法

マルチスレッドで実行されたプログラムについては，スレッド単位でメソッド呼び出し関係木を構築する．Java プログラムの場合は，スレッドオブジェクトに名前が与えられるため，そのスレッド名の対応関係により，同一名称のスレッドの呼び出し関係木の間で比較を行うものとした．スレッドの生成処理が変更されない限り，スレッドの名前は複数回の実行でほとんど変化がなく，著者らのこれまでいくつかのプログラムに対して分析を行ってきた経験の範囲では，スレッドの名前を用いた対応関係により問題を生じたことがない．

使用するスレッドの数が状況によって変わるようなプログラムではこの方式は採用できないため，スレッドの取りうる組に対してメソッド呼び出し関係木の比較を行い，一致するノード数が多いものを対応関係で結び付ける必要がある．それぞれのメソッド呼び出し関係木の根のノードから順番にメソッド呼び出しを比較していくアルゴリズムであるため，同一のスレッド生成処理で作られたスレッド間での比較以外には，スレッドの対応関係の計算のコストは，それほど大きな問題とはならないと推測している．

4. ケーススタディ

本手法によって，2 つの実行履歴の差分を検出し，可視化が行えているかを確認し，差分として検出された部分から実際にソースコードが変更を確認できるかを調べるために適用実験を行った．

4.1 実験対象

実験対象として画像編集ソフト JHotDraw[9] のバージョン 7.3 と 7.3.1 で同じシナリオを実行した実行履歴を取得した．

実験に使用したシナリオは以下の通りである．

(1) プログラムを起動する．
(2) 三角形描画のボタンをクリックする．
(3) キャンバスの中央でマウスをドラッグし三角形を描画する．
(4) Alt キーと F4 キーを同時に押してウィンドウを閉じる指示を出す．(Microsoft Windows において Alt+F4 はウィンドウを閉じるショートカットキーである．)
(5) 編集した図形データを保存するかどうかを問い合わせるダイアログが表示されるので,「保存しない」をクリックし，プログラムを終了する．

実行履歴を取得する際には，java，javax，sun のパッケージに含まれるライブラリへの呼び出しを除外した．

シナリオを実行し，取得した実行履歴の概要を表 1，2 に示す．表 1 では，取得した実行履歴全体のデータを，表 2 では，実行履歴のうちのスレッドごとのデータを示している．

4.2 実験方法

取得した実行履歴に Amida を改造して作成したツールを適用し，出現した実行履歴の差分から，実際にソースコードが変更されている箇所を特定できるか確認をする．

4.3 実験結果

2 つの実行履歴の比較を行い，差分の表示を行った．実行履歴の比較とシーケンス図の生成までの実行時間を調査したところ，およそ 11 分 30 秒かかった．

表 1 取得した実行履歴

バージョン	スレッド数	イベント数	ファイルサイズ (byte)
7.3	4	55,789	5,713,797
7.3.1	4	49,127	4,974,072

表 2 スレッドごとの実行履歴の内訳

バージョン	スレッド名	イベント数
7.3	main	21
	AWT-EventQueue	55,651
	Thread-3	2
	pool-1-thread-1	116
7.3.1	main	21
	AWT-EventQueue	48,989
	Thread-3	2
	pool-1-thread-1	116

2 つの実行履歴に含まれるスレッド数は共に 4 であり，同じスレッド名を持つスレッド同士で比較を行った．このとき，4 個のスレッドのうち，main，Thread-3，pool-1-thread-1 の 3 個のスレッドでは，全てのメソッド呼び出しが一致しており，差分は見つからなかった．スレッド AWT-EventQueue では，差分が検出されたので，このスレッドでの実行履歴の差分を参考に，ソースコードの変更点を探した．探す際に，実行履歴の差分の探索機能を使用したが，発見された差分は 300 箇所以上あったため，探索は途中で終了した．

その結果，実際に差分からソースコードの変更点を発見することができた．具体的には，org.jhotdraw.draw.AbstractTool クラスと org.jhotdraw.draw.DefaultDrawingEditor クラス，org.jhotdraw.draw.DefaultDrawingView クラスの 3 つである．

AbstractTool クラス

AbstractTool クラスでは，バージョン 7.3 から 7.3.1 に変わるときに，メソッド createActionMap の処理が変更された．具体的には，バージョン 7.3.1 ではメソッドの中の処理を全て削除し，null 値を返すだけになっていた．そのため，バージョン 7.3 では，メソッド createActionMap の呼び出しから他のメソッド呼び出しを行っていたが，バージョン 7.3.1 では，メソッド createActionMap が呼び出されてすぐに終了している．

DefaultDrawingEditor クラス

DefaultDrawingEditor クラスでは，バージョン 7.3 から 7.3.1 に変わるときに，新しくメソッドが 6 つ追加され，うち 2 つのメソッドがコンストラクタから呼び出されていた．そのため，コンストラクタ呼び出しで 2 つのメソッド呼び出しが増え，そのメソッド呼び出しが差分であると判断された．

図 5 は実際に生成したシーケンス図である．DefaultDrawingEditor クラスのコンストラクタ呼び出しの処理の一部であるが，メソッド setDefaultAttribute の呼び出しは変更されていないため強調表示されていないが，メソッド createInputMap とメソッド createActionMap の呼び出しは変更されているため，強調表示されている．またメソッド createActionMap では，さらに DeleteAction クラスのコンストラクタを呼び出しているが，メソッド createActionMap は新しく追加されたメソッドなので赤色に強調表示されている．

DefaultDrawingView クラス

DefaultDrawingView クラスでは，バージョン 7.3 と 7.3.1 の間でコンストラクタの呼び出しに変化があった．実際には，コンストラクタ内での処理は変わっていなかったが，クラス内のメンバ変数が変更されていた．そのため，コンストラクタの呼び出しが行われた際に，メンバ変数の初期化の処理がバージョン間で異なっていた．その動作の変化が，実行履歴に記録され，本手法で差分と判断された．

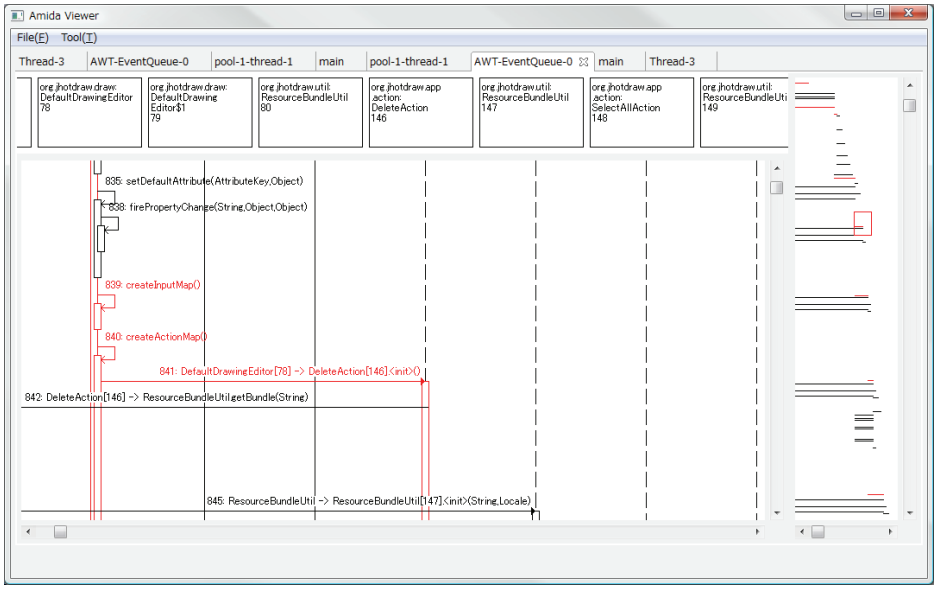

図 5 DefaultDrawingEditor クラスの変更による実行履歴の変化

その他の変化の検出

今回の適用実験では，多くの差分を検出し，そこから 3 つのクラスのソースコードの変更を発見したが，その他にもソースコードの変更とは関係がない実行履歴の差分を複数検出した．その原因はソースコードの変更が変更されていないクラスに影響を与えたことであった．具体的には IncreaseHandleDetailLevelAction クラスの初期化処理の動作が変わっていた．IncreaseHandleDetailLevelAction クラスのコンストラクタは DrawingEditor クラスが引数に指定されているが，バージョン 7.3 の実行時には DrawingEditorProxy クラスのオブジェクトが，バージョン 7.3.1 では，DefaultDrawingEditor クラスのオブジェクトが渡されていた．その結果，呼び出し先オブジェクトのクラス名が異なるため，差分と判断されていた．

4.4 考　　察

適用実験では，3 つのクラスで変更箇所を発見することができた．今回取得した実行履歴

は起動後に図形を 1 つ描画し終了するという簡単なものであったが，本手法によって，ソースコードの変更によるプログラムの振舞いの変化を検出できた．実験時には，差分を抽出したシーケンス図を主に使って調査したが，シーケンス図の内容のうち，動作が変化した部分を探す必要がなく，どのように変化したのかを調べることに注力することができた．強調表示されているとはいえ，巨大な 1 枚のシーケンス図から差分を探すにはそれなりの労力が必要であり，差分だけを別図として抽出する機能がなければ，調査は困難であったと思われる．

本手法は，実行履歴をシーケンス図として可視化するため，ソースコードの変更によってプログラムの振舞いにどのような影響が出たかを表示することができる．例えば，デバッグ作業のときに実装クラスを変更した場合，変更箇所が他のオブジェクトにどのような影響を与えるのかを図で示すことができる．ソースコードの変更は 1 箇所だけであっても，他の場所でプログラムの動作が変化するため，その影響を確認する必要がある．このような場合に本手法では，影響がある部分を変化部分として検出するため有効だと言える．

ソースコードが変更されていなくても，実行環境や入力データによるプログラムの振舞いの変化を本手法は検出することができる．入力データとシナリオを用意して実行し，実行履歴を取得すれば，入力データの違いによるプログラムの振舞いの違いは動作の変化として検出できる．バグを探す際には，入力の値を変更して実行し，動作を解析する必要があるため，複数の実行履歴を取得し，本手法を応用することでバグ検出に利用できると考えられる．

5. 関 連 研 究

本研究では，実行履歴の可視化手法と組み合わせることを前提とし，木構造の比較に基づく実行履歴の比較手法を提案している．実行履歴の比較手法として，Hoffman ら[8] は実行履歴をイベント系列とみなし，2 つのイベント系列の最長共通部分列を求める手法を提案している．最長共通部分列を求める手法は主に文章比較に使われ，2 つの文字列の中から最長となる共通部分を探索することで差分を見つける手法である．本研究における木構造の比較では，メソッド呼び出し関係木の根から比較するため，あるメソッド X から Y を直接呼び出すプログラムがメソッド X から Z を呼び，Z が Y を呼び出すように修正されたとき，この修正は「X からの呼び出しが変わった」と認識される．これに対して，イベント系列として比較した場合は，Y の実行そのものに対しては，対応関係を計算することが可能である．ただし，イベント系列の比較では，逆に，実際には無関係な呼び出し階層に偶然同じ系列が出現すると，それらを対応づける可能性もある．メソッド呼び出し関係木の比較から得られる差分は，必ずある 1 つのメソッドの実行が開始されてから終了するまでの単位として得られるため，出力が開発者にとって意味のある単位であり，シーケンス図等を用いた可視化手法との組み合わせが容易であるという利点がある．

Kuhn ら[10] は，実行履歴を，各時刻でのメソッド呼び出しのスタックの深さによって表現される数値の列として抽象化し，類似度の計算や可視化を行う手法を提案している．この手法を用いると，大規模な実行履歴に含まれている類似した実行系列を開発者が目視で確認することができる．一方で，本研究の目的である，デバッグにおけるプログラムの変更などによって生じる実行履歴のわずかな差異を検出する用途には不向きであると考えられる．

分析対象を GUI アプリケーションに絞った実行履歴の比較手法としては，Smit ら[16] は，ユーザの GUI 操作から起動された一連のメソッドを 1 機能だとみなし，その区間単位で実行履歴を対応づける手法を提案している．Smit らの手法もまた最長共通部分列を取り出す手法であるが，本研究にもこのアイディアを取り込むことは可能であり，それによって，実行シナリオ（ユーザの操作）の差異による影響を効果的に抽出できる可能性がある．

木構造の差分を求める手法は 2 つの木の編集距離やアライメント，包含関係を求める手法などが数多く研究されている[1]．木構造の差分を求める近似アルゴリズムに FMES[2] という手法がある．このアルゴリズムは比較する 2 つの木の間で，ノードのマッチング計算を行う．ノードを葉と内部ノードに区別し，類似度が閾値以下のノード同士はマッチしないという仮定を置いて計算する．n を比較する 2 つの木の葉の合計とし，e を木の差の大きさとすると計算量は $O(ne+e^2)$ になり，e が小さいほどよい近似解が得られる．デバッグ時のソースコードの変更は少なく，実行履歴の差分も小さいと考えられるため，この手法の条件に合致している．ただし，FMES では木構造の差分の近似解を求めるため，差分を正しく判断できない可能性がある．また，本手法のようにトップダウンに比較する SimpleTreePatternMatchingArgorithm[11] がある．この手法は二分木にしか適用できないが，文字列探索の Knuth-Morris-Pratt 法を利用した手法である．

本研究は，実行履歴の差分を開発者に提示することを目的としているため，実行履歴から作られた木は一致しているか，あるいは不一致であるかのどちらかである．一方，プログラムの実行時の動作を監視して異常を検出する，欠陥の位置を特定するなどの研究では，入力データや環境のわずかな差異を吸収して，同じ実行履歴として扱うための手法も様々に研究されている．たとえば，Reiss [13] は，システムの性能を計測することを目的として，10 ミリ秒の時間単位で呼び出されたメソッドの数や確保されたオブジェクトの数を集計したベクトルを作成しておき，ベクトルの変化によってプログラムの動作の変化を検出する手法を提案している．また，Diep らは，欠陥の位置を自動特定する手法において意味がないと思わ

れる動作の差異を取り除くために，実行した文のカバレッジが等しいような条件分岐による差異はまとめてしまうという手法 [5] と，メソッド呼び出しの順序が異なっても結果として得られる状態が同じであれば動作は同じとみなして呼び出し順序を正規化する手法 [6] を提案している．

6. ま と め

本研究では，プログラムの変更前後での実行履歴の差分検出手法を提案した．提案した手法はデバッグ時のソースコードのわずかな変更によって生じる実行時のメソッドの呼び出し関係の変化を検出し，可視化して表示する．バージョンの異なるオープンソースソフトウェアを対象にしたケーススタディにより，バージョン間の変更点を発見することに成功した．

今後の課題として，木構造の比較にかかる時間の削減が挙げられる．実行履歴はプログラムの実行から終了までの全てのメソッド呼び出しを取得をするため巨大になりがちであるが，本手法ではメソッド呼び出しの数が多くなるほど比較計算に時間が多くかかってしまう．そこで木構造の比較アルゴリズムを改良し，計算コストを減らすことを考えている．例えば，前処理として部分木のハッシュ値を計算することで比較計算にかかる時間を削減できる．また，Amida のループ圧縮機能を利用し，木構造に含まれる繰り返し処理によるノードを事前に減らすことで比較計算にかかる時間を削減する手法を検討している．

謝辞 本研究は，文部科学省科学研究費補助金若手研究（B）（課題番号:21700030）の助成を得た．

参 考 文 献

1) P.Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, Vol. 337, pp. 217–239, June 2005.
2) S.S. Chawathe, A.Rajaraman, H.Garcia-Molina, and J.Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 493–504, 1996.
3) H.Cleve and A.Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pp. 342–351, 2005.
4) J.K. Czyz and B.Jayaraman. Declarative and visual debugging in eclipse. *Eclipse Technology Exchange*, 2007.
5) M.Diep, S.Elbaum, and M.Dwyer. Reducing irrelevant trace variations. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 477–480, 2007.
6) M.Diep, S.Elbaum, and M.Dwyer. Trace normalization. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pp. 67–76, 2008.
7) M.Hamill and K.Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering*, Vol.35, No.4, pp. 484–496, 2009.
8) K.J. Hoffman, P.Eugster, and S.Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 453–464, 2009.
9) JHotDraw. `http://sourceforge.net/projects/jhotdraw/`.
10) A.Kuhn and O.Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 320–329, 2006.
11) H.T. Lu and W.Yang. A simple tree pattern-matching algorithm. In *Proceedings of the Wrokshop on Algorithm and Theory of Computation*, December 2000.
12) W.D. Pauw, D.Lorenz, J.Vlissides, and M.Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems*, pp. 219–234, April 1998.
13) S.P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the 3rd International Workshop on Dynamic Analysis*, pp. 1–6, 2005.
14) S.P. Reiss and M.Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pp. 221–230, May 2001.
15) T.Richner and S.Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of the 18th International Conference on Software Maintenance*, pp. 34–43, October 2002.
16) M.Smit, E.Stroulia, and K.Wong. Use case redocumentation from gui event traces. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pp. 263–268, 2008.
17) Unified ModelingLanguage(UML)1.5 specification, March 2003. OMG.
18) T.Systä, K.Koskimies, and H.Müller. Shimba - an environment for reverse engineering java software systems. *Software Practice and Experience*, Vol.31, No.4, pp. 371–394, 2001.
19) N.Wilde and R.Huitt. Maintenance support object-oriented programs. *IEEE Transactions on Software Engineering*, Vol.18, No.12, pp. 1038–1044, 1992.
20) 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラムの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.