修士学位論文

題目

木構造の比較に基づくメソッド呼び出し履歴の変化の可視化手法

指導教員
井上 克郎 教授

報告者
伊藤 芳朗

平成 22 年 2 月 8 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 21 年度 修士学位論文

木構造の比較に基づくメソッド呼び出し履歴の変化の可視化手法

伊藤 芳朗

内容梗概

オブジェクト指向プログラムでは，実行時に動的に生成されるオブジェクトが相互にメッセージを交換する事によってシステムが動作する．そのため，システムの振る舞いを理解するには，ソースコードなどの静的な情報を解析するだけでなく，メソッド呼び出し履歴などの動的な情報を解析する手法が有効である．開発者はソースコードを変更したあとにプログラムの振る舞いの変化が意図したものであるのかを確かめる必要がある．プログラムのメソッド呼び出し履歴は実行時の動作を記録したものであるため，ソースコードが変更された場合，プログラムの振る舞いが変化し，プログラムの実際の動作を記録したメソッド呼び出し履歴も変化する．そこで，ソースコードを変更した前後で同じシナリオを実行したメソッド呼び出し履歴を比較し，違いを検出，可視化することで，ソースコードの変更による振る舞いの変化を確認することができる．しかし，メソッド呼び出し履歴は膨大な量になることが多く，メソッド呼び出し履歴の内容の対応関係を人の手で調査し，振る舞いが変化した部分を特定するのは現実的ではない．

そこで本研究では，ソースコードを変更した前後で同じシナリオを実行したメソッド呼び出し履歴からプログラムの振る舞いの変化を可視化する手法を提案する．メソッド呼び出し履歴間の変化を検出し可視化することで，開発者はプログラムの振る舞いの変化が意図したものであるかを判断することが可能になる．具体的には，取得したメソッド呼び出し履歴からメソッド呼び出し間の関係を取得し，これを比較することで 2 つのメソッド呼び出し履歴間の違いを検出する．メソッド呼び出し履歴を可視化する際には，利用者が分かりやすいようにメソッド呼び出し履歴間の違いを強調表示することで，プログラムの振る舞いの変化を確認しやすくする．加えて，メソッド呼び出し履歴全体から振る舞いが変化した部分だけを抽出して可視化することもできる．

本手法の評価実験として，提案手法を実装したシステムを作成し，2 つの異なるバージョンの画像編集ソフトから同じシナリオを実行したメソッド呼び出し履歴を取得し適用した．その結果，表示されたメソッド呼び出し履歴の変化から 2 つのバージョン間で変更が行われた箇所を特定できることを確認した．

主な用語

プログラム理解
メソッド呼び出し履歴
木構造の比較
差分検出
可視化

目 次

1 まえがき **4**

2 背景 **6**

2.1 オブジェクト指向プログラム 6
2.2 UML シーケンス図 6
2.3 Amida 8
2.4 メソッド呼び出し履歴の変化の要因 9
2.5 メソッド呼び出し履歴の比較 10

3 提案手法 **11**

3.1 メソッド呼び出し履歴の取得 11
3.2 メソッド呼び出し関係木の構築 12
3.3 メソッド呼び出し履歴の変化の検出 13
3.4 メソッド呼び出し履歴の変化の可視化 16

4 実装 **18**

4.1 メソッド呼び出し履歴の変化部分の探索と抽出表示 19

5 適用実験 **20**

5.1 実験目的 20
5.2 実験対象 20
5.3 実験方法 21
5.4 実験結果 22
5.5 考察 24

6 関連研究 **26**

7 まとめ **28**

謝辞 **29**

参考文献 **30**

1　まえがき

オブジェクト指向プログラムでは，実行時に動的に生成されるオブジェクトが相互にメッセージを交換する事によってシステムが動作するが，どのオブジェクトがどのようにメッセージ通信を行うかは，実行時に動的に決定される．そこで，このようなシステムの振る舞いを理解するためには，ソースコードの解析を行うだけでなく，プログラムの実行を観測し，実際の振る舞いを可視化することが有効である [18]．プログラムの実際の振る舞いを観測した記録をメソッド呼び出し履歴と呼ぶ．

ソースコードに変更を加えると，プログラムの振る舞いが変化する．例えばプログラムのバグを発見したとき，バグを修正するためにソースコードに変更を加える．このとき開発者はソースコードの変更によるプログラムの振る舞いの変化が意図したものであるのかを確かめる必要がある．そこで，ソースコードを変更した前後で同じシナリオを実行したメソッド呼び出し履歴を比較すれば，ソースコードの変更による振る舞いの変化が分かる．しかし，メソッド呼び出し履歴は膨大な量になることが多く，人の手で振る舞いの変化を探すのは現実的ではない．

本研究では，ソースコードを変更した前後で同じシナリオを実行してメソッド呼び出し履歴を取得し，そのメソッド呼び出し履歴の差分を検出する手法を提案する．検出したメソッド呼び出し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る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，取得したメソッド呼び出し履歴からメソッド呼び出し関係を木構造で表し，2 つのメソッド呼び出し履歴の比較を行うことでメソッド呼び出し履歴の変化を検出する．メソッド呼び出し履歴を可視化する際には，利用者がわかりやすいようにメソッド呼び出し履歴の変化部分を強調表示する．また，メソッド呼び出し履歴の変化を抽出することでプログラムの振る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．デバッグ作業は何らかのバグを発見した際に，バグを特定し修正する作業であり，バグを修正したあとに修正箇所のテストを行い，プログラムが意図したとおりに修正されたかを確認する．このときに本手法を適用し，メソッド呼び出し履歴の変化を可視化することで，デバッグによるプログラムの振る舞いの変化を発見することができる．また，メソッド呼び出し履歴の変化となる部分の探索を行うことで，デバッグ担当者の意図しない振る舞いの変化を確認できる．

我々の研究チームは，メソッド呼び出し履歴からシーケンス図を生成するシステム Amida [19] を開発している．Amida は，プログラムの動作の理解支援を目的として，プログラム実行の観測から得たメソッド呼び出し履歴を基にシーケンス図の作成を行う．本研究ではこの Amida

を改造し，メソッド呼び出し履歴からシーケンス図を生成する過程で，メソッド呼び出し履歴の変化を検出する機能と変化部分を強調表示する機能とシーケンス図上に存在する変化部分を抜き出して表示する機能を実装した．そして，実際に画像編集ソフト JHotDraw [7] の 2 つのバージョンで同じシナリオを実行したメソッド呼び出し履歴を取得し，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変更が行われた箇所を特定できることを確認した．

以降，2 章ではこの研究の背景を述べ，3 章で提案手法の説明をする．4 章では実装について説明し，5 章では行った適用実験とその結果について述べる．6 章では関連研究を紹介する．最後に 7 章で本研究の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．そのため，このようなシステムの振る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だけでなく，プログラムのメソッド呼び出し履歴のような動的な情報を解析する手法が有効である．

メソッド呼び出し履歴はプログラム実行時に呼び出されたメソッド呼び出しイベントの系列である．単純な調査方法は，呼び出し情報をテキスト形式で書き出しておき，開発者が閲覧するというものであるが，それだけではオブジェクト間の相互作用を理解するのは難しい．一方で，設計段階においては，オブジェクトの相互作用は主に UML のシーケンス図によって記述される．シーケンス図によって表現される動作シナリオは，そのソフトウェアにおける特定のタスクを実現する方法を記述しており，ソフトウェアの理解や再利用にも適している [14]．そのため，プログラムのメソッド呼び出し履歴からシーケンス図を生成する手法が提案されている [3, 17, 19]．

2.2 UML シーケンス図

シーケンス図とは Unified Modeling Language(UML) [16] 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，時系列に沿って示すことができる図である．横軸はオブジェクトの種類を表しており，図の上部には，図中に記述されるメッセージ通信に関連するオブジェクトが横方向に並べられる．図 1 に示す例では，クラス A のオブジェクト 1，クラス B のオブジェクト 2，というように 5 つのオブジェクトが配置されている．縦軸は時間軸を表しており，下方に行くほど時間が経過していく．各オブジェクトの下部には縦方向に点線が引かれており，これがオブジェクトが生存する区間を示している．さらに，個々のメッセージ通信について，時系列順に，送信元のオブジェクトから，送信先のオブジェクトに対して矢印を引く．送信されたメッセージがメソッド呼び出しだった場合は，メッセージを受けたオブジェクトは，そのメソッドの実行区間を縦長の長方形で表す．メソッドが終了する時点で，呼び出し元のオブジェクトへ戻り辺の矢印を引く．メッセージがオブジェクトの生成だった場合は，生成されるオブジェクトをその高さに書く．図 1 では，オブジェクト 1 に対するメソッド a の呼び出しが起き，プログラムの実行中にオブジェクト 2 に対するメソッド b の呼び出しが起きている．また，メソッド b の実行中にもオブジェクト 3 に対する呼び出しが起きている．このようにして，オブジェクト間のメッセージの様子を時系列に沿って表現する．

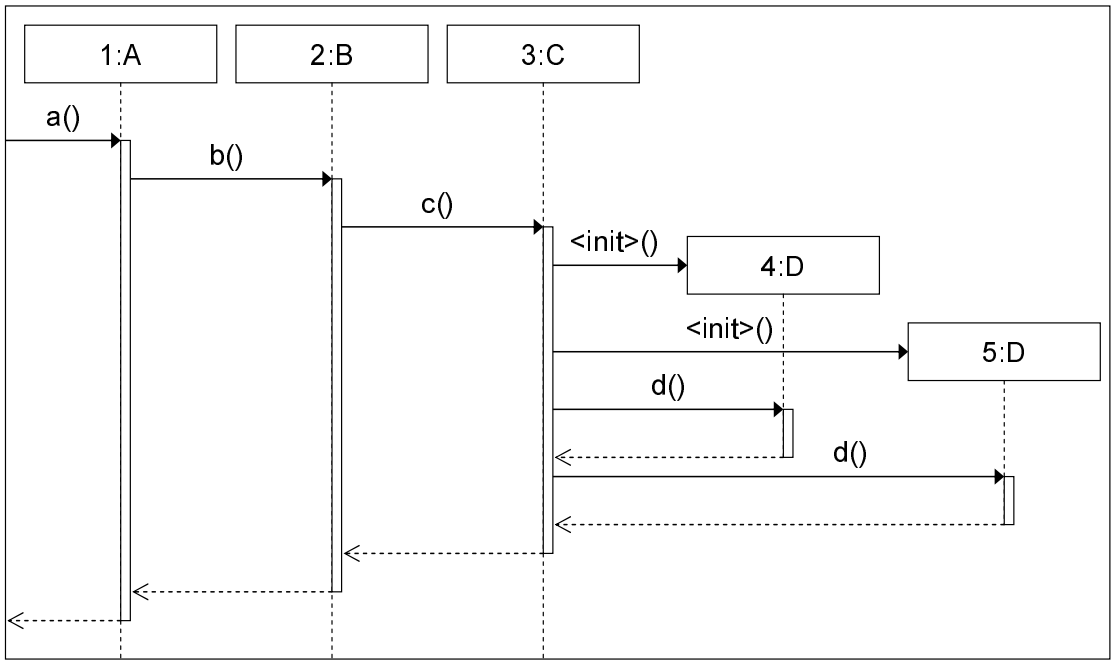

図 1: シーケンス図

UML で書かれた振る舞いの情報は，それ自体もプログラムの動作を理解するために有益であるが，設計図を活用することが困難な場合もある．たとえば，開発が進行していく中でソフトウェアの機能に変更が加えられたとき，開発者が設計図の更新を怠ると，設計図がソフトウェアの最新の状態を正しく反映しなくなる [9]．また，ソフトウェアを実装していく段階で，設計図には登場しないようなクラス，メソッドが追加されることもある [19]．

そこで，開発されたソフトウェアからオブジェクト間の相互作用を検出し，UML のシーケンス図を含む様々な形式によって可視化する方法が研究されている [17]．ソースコードの解析によって得られる結果は，ソースコードから読み取れる範囲でしか動的束縛などを判断できないため，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メソッド呼び出し履歴を解析し，可視化する手法が広く研究されている．

メソッド呼び出し履歴は，プログラムの実行開始から終了までの，膨大な数のメソッド呼び出しの系列である．例えば，適用実験で使用している JHotDraw7.3 では，プログラムを起動してから三角形の図を 1 つ描画し終了するという動作で，5 万回を超えるメソッド呼び出しとなっている．シーケンス図として可視化するためには，メソッド呼び出し履歴として各メソッド呼び出しのオブジェクト ID とメソッド名を記録する必要がある [19]．

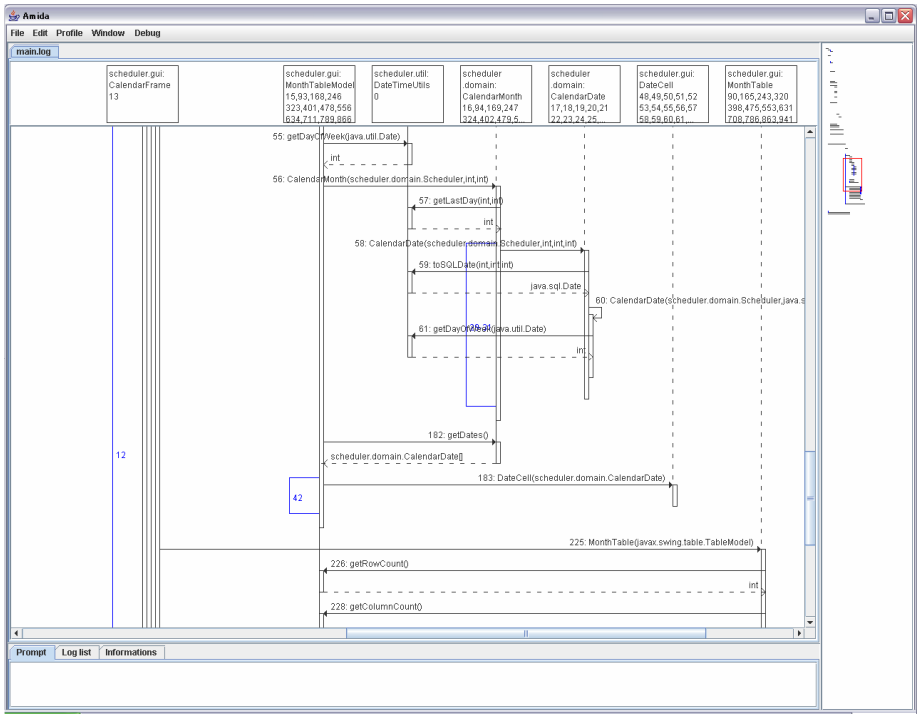

図 2: シーケンス図生成システム Amida

2.3 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る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メソッド呼び出し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 [19]．Amida は Java プログラムのメソッド呼び出しをメソッド呼び出し履歴として取得するプロファイラと，そのメソッド呼び出し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 GUI の図で，図の左側にシーケンス図を表示する．ただし，メソッド呼び出し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の表している．

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．ループの部分を実際に圧縮して表示すると，図 3 のように表示される．図 3 の 26 という数字は，getDaiBunrui，getChuBunrui，getShoBunrui という 3 つのメソッドがこの順に 26 回繰り返していることを表している．

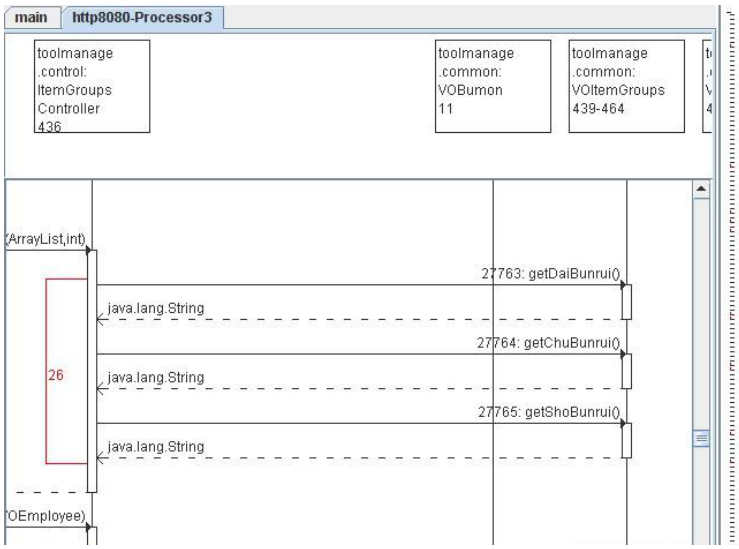

図 3: ループの圧縮

2.4 メソッド呼び出し履歴の変化の要因

プログラムのメソッド呼び出し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メソッド呼び出し履歴が変化する場合がある．その要因は次の 3 つに分類できる．

- ソースコードの変更
- 実行環境の変化
- 入力データの変化

ソースコードが変更された場合は，変更に応じてプログラムの振る舞いが変わるため，メソッド呼び出し履歴が変化する．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メソッド呼び出し履歴に追加される．また，動的束縛があるためメソッドのオーバーライドなどを行うと呼び出し側のコードが変更されていなくとも，動作が変わることもある．

実行環境が変化した場合は，実行環境ごとにメソッド呼び出し履歴が変化する．例えば，OS ごとに処理を変更する場合がある．その場合，ソースコードや入力の値が全く同じでも，メソッド呼び出し履歴が変化する．また，スレッドの実行などの非同期の処理は実行ごとに実行環境によっても大きく変化することがある．

入力データが変化した場合は，入力データの値によってメソッド呼び出し履歴が変化する．例えば，外部ファイルを読み込むプログラムの場合，ファイル内にあるデータの数によって

読み込みを行うときのループの回数が変わることがある．また，起動時に設定ファイルを読み込むプログラムの場合，設定ファイルが失われていると読み込みに関する処理は全く行われなくなる．

同じ実行をした 2 つのメソッド呼び出し履歴を比較した場合，メソッド呼び出し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る舞いの変化の要因を発見できると考えられる．

2.5 メソッド呼び出し履歴の比較

プログラムの開発者がバグを発見したとき，バグを修正するためにソースコードの修正を行う．このとき開発者は単体テストなどを行い，修正が正しく行われたか，また他の場所で意図しない変化が起きていないかを確かめる必要がある．デバッグ作業でのソースコードの変更によってプログラムの振る舞いが変化すれば，メソッド呼び出し履歴も変化する．そこで，ソースコードを変更した前後で同じシナリオを実行したメソッド呼び出し履歴を比較し，変更によるプログラムの振る舞いの変化を検出することで，ソースコードの変更による振る舞いの変化が分かる．

ただし，メソッド呼び出し履歴はプログラムの開始から終了までの全てのメソッド呼び出しを記録するため膨大になりがちという問題点がある [11, 13, 14]．さらにデバッグ作業ではソースコードの変更箇所が少ないことが考えられるため，メソッド呼び出し履歴の変化もメソッド呼び出し履歴全体の中でごく一部にしか現れない可能性がある．そのため，人の手によって膨大なメソッド呼び出し履歴同士を比較し，変更によるプログラムの振る舞いの変化が正しいのかを判断するのは困難である．

また，計算機を利用してメソッド呼び出し履歴を比較する場合でも，問題点がある．メソッド呼び出し履歴のオブジェクトの ID はプログラム実行時の生成順序やメモリ上の配置などを元に実行ごとに決定されるため，同じシナリオを実行して取得したメソッド呼び出し履歴でも生成したオブジェクトの ID が異なるという特徴がある．そのため，オブジェクトの ID を利用して同じ役割をもつオブジェクトの対応を取ることができない．

3　提案手法

本研究では，2つのメソッド呼び出し履歴を比較し，メソッド呼び出し履歴の変化の位置を特定する手法を提案する．特定したメソッド呼び出し履歴の変化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る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，取得したメソッド呼び出し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，木構造に変換する．そして，ソースコードの変更前後の木構造のメソッド呼び出し履歴の比較を行い，その違いとなる部分を求めることでメソッド呼び出し履歴の変化を検出する．

検出したメソッド呼び出し履歴の変化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，可視化する際にメソッド呼び出し履歴の変化した部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，メソッド呼び出し履歴全体をシーケンス図として生成し，メソッド呼び出し履歴の変化となる部分を強調して表示する．また，メソッド呼び出し履歴が膨大になることを考慮し，可視化したメソッド呼び出し履歴の中から振る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 プログラムのメソッド呼び出し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 プログラムのメソッド呼び出し履歴に対応した手法となっているが，同様な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　メソッド呼び出し履歴の取得

提案手法が対象とするメソッド呼び出し履歴は，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．具体的には，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．

Amida のメソッド呼び出し履歴を取得するシステムは，対象とするプログラムの実行中に個々のメソッド呼び出しを捉え，図 4 の下部に示す形式でメソッド呼び出し履歴をファイ

ルに保存する．タイムスタンプは整数の値をとり，プログラムの開始から 1 回の呼び出しごとに 0 から値が増えていく．メソッドの終了時には「@スレッド番号 }」のみを記録し，それぞれ対応する開き括弧のメソッドの終了を表している．static メソッドの呼び出しについては，オブジェクト ID を 0 番とする．本研究では，木構造を構築するときにはこれらの全ての情報を保持させる．

JHotDraw の実行から取得したメソッド呼び出し履歴の例を図 4 に示す．この図は JHotDraw の Main クラスの main メソッドを実行したときのメソッド呼び出し履歴の一部である．図のメソッド呼び出し履歴の 1 行目のメソッド呼び出しを見ると以下の情報が分かる．

- スレッド番号: 1
- タイムスタンプ: 0
- 返り値の型: void
- クラス名: org.jhotdraw.samples.draw.Main
- オブジェクト ID: 0
- メソッド名: main
- 引数の型: java.lang.String[]

3.2 メソッド呼び出し関係木の構築

最初に，取得したメソッド呼び出し履歴をメソッドの呼び出し関係を元に木構造に変換する．メソッド呼び出し履歴は，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．このとき，あるメソッド M_1 が別のメソッド M_2 を呼び出す場合，メソッド呼び出し履歴の中では，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メソッド呼び出し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．メソッド呼び出し履歴中で，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には意味があるため，メソッド

```
@1 0 void org.jhotdraw.samples.draw.Main(0).main(java.lang.String[]){
@1 1 void org.jhotdraw.app.DefaultSDIApplication(1).<init>(){
@1 2 void org.jhotdraw.app.AbstractApplication(1).<init>(){
@1 3 void org.jhotdraw.beans.AbstractBean(1).<init>(){
@1 }
@1 }
@1 }
@1 4 void org.jhotdraw.samples.draw.DrawApplicationModel(2).<init>(){
@1 5 void org.jhotdraw.app.DefaultApplicationModel(2).<init>(){
@1 6 void org.jhotdraw.beans.AbstractBean(2).<init>(){
@1 }
@1 }
@1 }
@1 7 void org.jhotdraw.app.DefaultApplicationModel(2).setName(java.lang.String){
```

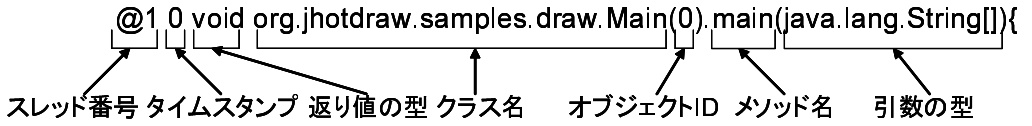

図 4: メソッド呼び出し履歴の例

呼び出し関係木は順序木とし，メソッド M_2 とメソッド M_3 のノードは兄弟ノードであり，メソッド M_2 を兄ノードとする

例えば，図 4 で示したメソッド呼び出し履歴をメソッド呼び出し関係木にすると，図 5 のようになる．図 5 のノードの数字は図 4 のタイムスタンプに対応している．まず最初にタイムスタンプが 0 のメソッド呼び出しが起きるため，根としてノード 0 が作られる．次にタイムスタンプが 1 から 3 までのメソッド呼び出しが起きるため，そのメソッド呼び出しのノードはそれぞれ親子関係になり，メソッド呼び出し関係木に追加される．次にタイムスタンプが 4 であるメソッド呼び出しが起きるが，それまでにメソッドの終了が 3 回起きているため，ノード 4 はノード 0 の子ノードとなる．このときに先に呼ばれたメソッド呼び出しが左側に来るように子ノードを追加する．以降も同様に処理を行うと図 5 のメソッド呼び出し関係木が得られる．

3.3 メソッド呼び出し履歴の変化の検出

次に，2 つのメソッド呼び出し履歴から得られた 2 つのメソッド呼び出し関係木の比較を行う．メソッド呼び出し関係木の比較を行い，2 つのメソッド呼び出し関係木の中に一致する部分があれば，その部分は変化していないことになる．そのため，ノードには一致する部分があるかどうかをマークできるように領域を確保する．

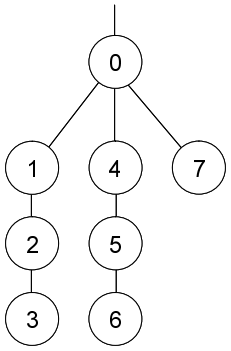

図 5: メソッド呼び出し関係木

ソースコードの変更や実行環境の変化，入力データの変化などの要因により，メソッド呼び出し履歴は変化するが，本手法では，ソースコードの変更を主に想定する．その場合，2 つのメソッド呼び出し履歴の間には大きな違いがないと考えられる．そこで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，木の根から順に比較を行っていく．

メソッド呼び出し関係木の比較のときに，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．

また，ノード r_a と r_b が等しい条件を以下のように設定する．

- ノード r_a と r_b の呼び出し元オブジェクトのクラス名が等しい．
- ノード r_a と r_b の呼び出し先オブジェクトのクラス名が等しい．
- ノード r_a と r_b のメソッド呼び出し名が等しい．
- ノード r_a と r_b の引数の数と型が全て等しい．
- ノード r_a と r_b の返り値の型が等しい．

なお，クラス名や型を比較する際は完全限定名を使用する．

メソッド呼び出し履歴を取得した際に，メソッド呼び出しイベントには呼び出し元オブジェクト名は記録されていない．しかし，メソッド呼び出し間の関係がメソッド呼び出し関係木の親子関係となるため，呼び出し元オブジェクトのクラス名はそのノードの直接の親ノードの呼び出し先オブジェクトのクラス名である．そのため各ノードには親となるノードへのポインタを保持させる．

ノードの一致にオブジェクト ID を使用しないのは，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．このため，オブジェクト ID の比較によって，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 6 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．このとき，2 つのノードの子ノードの数が等しいため，i 番目の子ノードを根とする部分木が等しいかの判定を行う．1 番目の子ノードは共に 2 なので等しい．ノード 2 は共に子ノードを持たないため，ノード 2 を根とする部分木は等しく，ノード 1 の 1 番目の子ノードを根とする部分木は等しい．2 番目の子ノードも共に 3 なので等しい．そのため，ノード 3 を根とする部分木が等しいかを同様に判定する．ここでもノード 3 を根とする部分木は等しいため，ノード 1 の 2 番目の子ノードを根とする部分木は等しい．その結果全ての子ノードを根とする部分木が等しいため，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．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，メソッド呼び出し履歴の変化部分であると分かる．図 6 では，例を挙げた部分の他に一致する部分木がないため，マークするのはノード 1 を根とする部分木だけである．

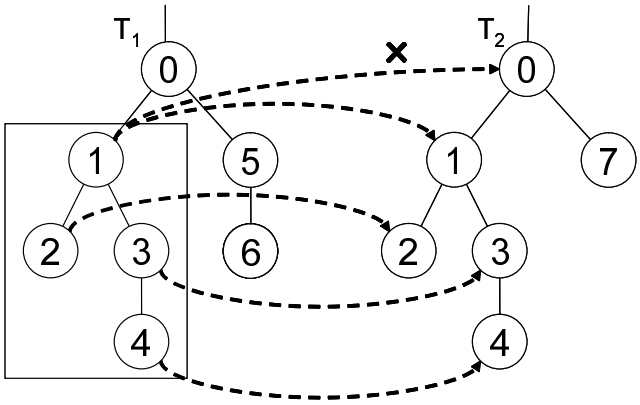

図 6: メソッド呼び出し関係木の比較

3.4 メソッド呼び出し履歴の変化の可視化

メソッド呼び出し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，メソッド呼び出し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 [19] 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 7 は図 5 をシーケンス図にしたものである．図 5 のノードのタイムスタンプを図 7 のメソッド呼び出しを表す矢印に添えた．あるノードが直接の子ノードを持つ場合，そのノードのメソッド呼び出しの実行中に左側から順にメソッド呼び出しを記述する．

また，図 8 は図 6 にメソッド呼び出し関係木で示した 2 つのメソッド呼び出し履歴をそれぞれシーケンス図として可視化し，メソッド呼び出し履歴上の変化を強調表示したものである．図 8 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 6 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 以下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 8 のように強調表示を行う．

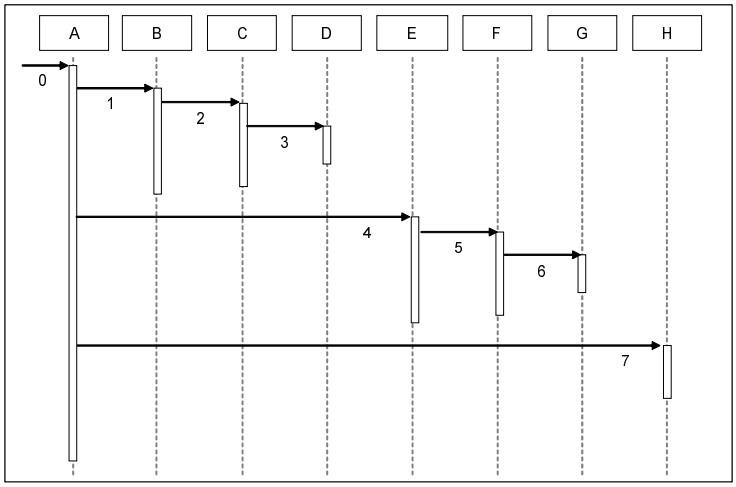

図 7: シーケンス図生成の例

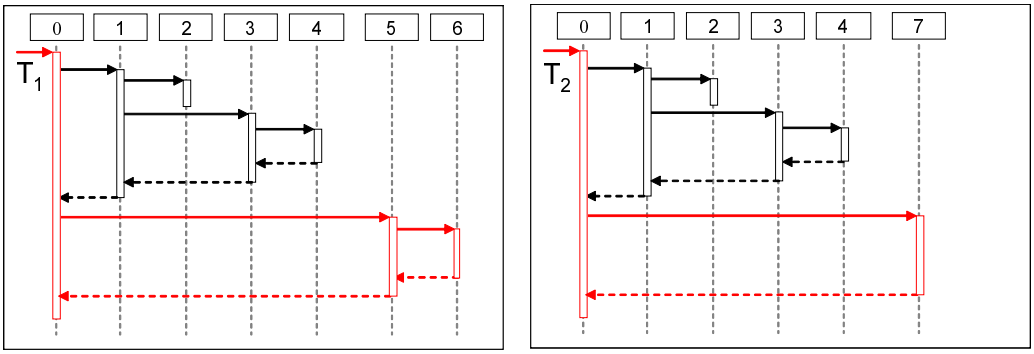

図 8: シーケンス図の比較

4 実装

3 章で述べた手法を，2.3 節で紹介をした Amida [19] を改造して実装した．Amida は Java プログラムを対象とし，プログラムのメソッド呼び出し履歴を取得する機能と，それを解析してシーケンス図を生成し，GUI にてシーケンス図を表示する機能を持つ．メソッド呼び出し履歴の取得には Amida の機能をそのまま使用する．メソッド呼び出し履歴の変化を表示するために Amida のシーケンス図生成と表示を行う機能を改造することで実装を行った．

今回 Amida に追加する処理は以下の 3 つである．

- メソッド呼び出し履歴の比較計算
- メソッド呼び出し履歴の変化部分の強調表示
- メソッド呼び出し履歴の変化部分の探索と抽出

Amida はシーケンス図を生成する際に，取得したメソッド呼び出し履歴を木構造に変換する．Amida ではこれをコールツリーと呼ぶが，コールツリー内のノードに保持されるメソッド呼び出し履歴の情報は，本手法で必要とする情報を全て保持しているため，ノードに一致する部分木がある際にマークする領域を追加し，実装では Amida のコールツリーを使用した．また，Amida は GUI 上でスレッド単位でシーケンス図を表示するため，スレッド単位でコールツリーに変換する．そのため，本手法も同様にスレッド単位でメソッド呼び出し関係木を構築する．Amida がメソッド呼び出し履歴を取得する際には，スレッドの開始時にスレッド名を記録することを利用し，同じスレッド名を持つメソッド呼び出し関係木同士を比較する．

Amida はメソッド呼び出し履歴をコールツリーに変換し，それを元にシーケンス図を生成する．そのため，2 つのメソッド呼び出し履歴を入力するように変更し，比較計算はコールツリーの構築とシーケンス図の生成を行う間に処理を行う．シーケンス図の生成時には，コールツリーは比較計算の結果を保持しているため，シーケンス図を生成する際に，一致する部分がない場合は強調表示を行う．ただし，Amida は GUI 上でシーケンス図中のメソッド呼び出しやオブジェクトを選択した場合，関連するメソッド呼び出しやオブジェクトを青色で表示する機能を持つ．そのため，描画時の色の選択は，まずユーザが描画するメソッド呼び出しを選択しているかどうか，次に描画するメソッド呼び出しがメソッド呼び出し履歴の変化部分であるかどうかで色の選択を行う．これは GUI の右側にあるシーケンス図の全体図でも同様に行う．

4.1 メソッド呼び出し履歴の変化部分の探索と抽出表示

Amida には表示しているシーケンス図から一部を抽出して個別のシーケンス図を生成する機能が実装されているため，それを利用してメソッド呼び出し履歴の変化部分の抽出表示を行う．Amida ではスレッドごとにシーケンス図を生成するが，GUI 上ではタブを使って表示するスレッドを選択している．このときシーケンス図から抽出した一部のシーケンス図を表示する際には新しいタブが作られ，そこを選択することで表示できる．そのため，変化部分が多い場合に，大量のタブができすぎないように，メソッド呼び出し履歴の変化部分の探索は指定したスレッドのみを対象として行う．それでも 1 つのシーケンス図のサイズが巨大であれば，変化部分も膨大な数になる可能性がある．そこで，変化部分が 300 箇所発見された場合，それ以上の探索を中止し，300 箇所まで表示するようにした．抽出したあとのシーケンス図でも強調表示は同様に行う．

指定したスレッドの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メソッド呼び出し履歴の変化となる部分を探索する．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，該当したメソッド呼び出しの親となるオブジェクトへのメソッド呼び出しと，そのオブジェクトが呼ぶ全てのメソッド呼び出しを表示する．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．

図 9 は実装したツールのスクリーンショットである．図は変化部分を探索し抽出表示を行ったときのものである．図中のメソッド createActionMap と setActionMap で変化があるため，それに関連した部分を抽出し，図のように赤色で強調表示されている．抽出したシーケンス図には関連したオブジェクトしか表示されないため，図の上部にはオブジェクトが 3 つしか並んでいない．もし図に無関係なオブジェクトが大量に並んでいると，メソッド呼び出しを表す矢印はとても長くなり，一画面に収まらなくなってしまう．また，シーケンス図の縮小図も関連した呼び出しだけを取り出して表示するため，とても簡単なものになっている．

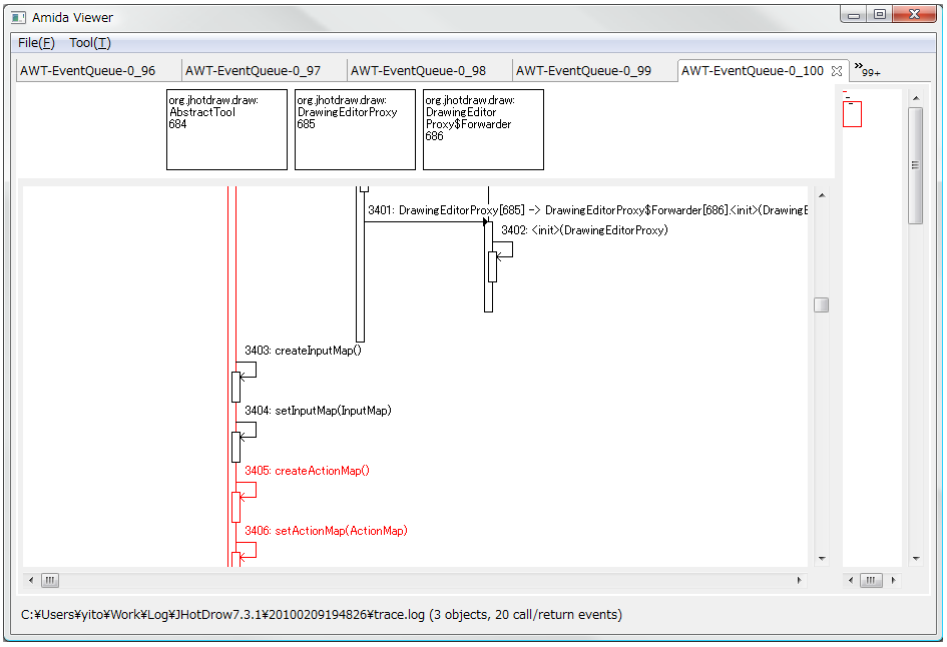

図 9: メソッド呼び出し履歴の変化部分の抽出表示

5 適用実験

4 章で述べたツールを用いて適用実験を行った．本章ではその内容および結果と結果に対する評価を述べる．

システムの実行環境は以下の通りである．

- CPU: Intel Core2 Quad CPU 2.40GHz
- RAM: 2.00GB
- OS: Mictrosoft Windows Vista Ultimate Service Pack2

5.1 実験目的

本手法によって，2 つのメソッド呼び出し履歴の差分を検出し，可視化が行えているかを確認する．また，本手法で差分として表示される部分からソースコードを特定し，実際にソースコードが変更されているかを確認する．

5.2 実験対象

実験対象として画像編集ソフト JHotDraw [7] のバージョン 7.3 とバージョン 7.3.1 で同じシナリオを実行したメソッド呼び出し履歴を取得した．

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．)
    - すると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．
5. 保存しないをクリックし，プログラムを終了する．

メソッド呼び出し履歴を取得する際には，以下のパッケージに含まれるライブラリをフィルタし，それ以外のメソッド呼び出し履歴を取得した．

- java.*
- javax.*
- sun.*

シナリオを実行し，取得したメソッド呼び出し履歴の概要は表 1，2 のようになった．表 1 では，取得したメソッド呼び出し履歴全体のデータを，表 2 では，メソッド呼び出し履歴のうちのスレッドごとのデータを示している．

5.3 実験方法

取得したメソッド呼び出し履歴に 4 章で作成したツールを適用し，出現したメソッド呼び出し履歴の変化部分から，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．

表 1: 取得したメソッド呼び出し履歴

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

5.4 実験結果

2 つのメソッド呼び出し履歴の比較を行い，変化となる部分の表示を行った．メソッド呼び出し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．

2 つのメソッド呼び出し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えていた．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，変化となる部分は見つからなかった．スレッド AWT-EventQueue では，変化した部分が検出されたので，このスレッドでのメソッド呼び出し履歴の変化を参考に，ソースコードの変更点を探した．探す際に，メソッド呼び出し履歴の変化部分の探索機能を使用したが，発見された変化部分は 300 箇所以上あったため，探索は途中で終了した．

その結果，実際に変化部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 からバージョン 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．

表 2: スレッドごとのメソッド呼び出し履歴の内訳

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

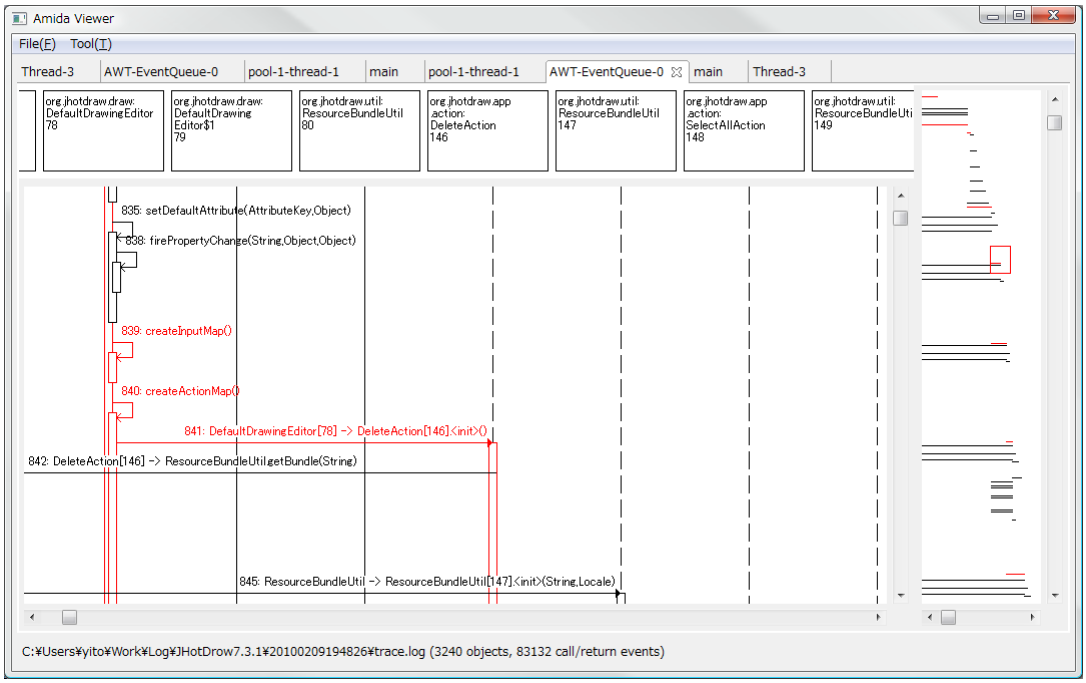

図 10: DefaultDrawingEditor クラスの変更によるメソッド呼び出し履歴の変化

DefaultDrawingEditor クラス

DefaultDrawingEditor クラスでは，バージョン 7.3 からバージョン 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変化部分であると判断された．

図 10 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た部分であるため，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 とバージョン 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，メソッド呼び出し履歴に記録され，本手法で変化部分であると判断された．

その他の変化の検出

今回の適用実験では，多くの変化部分を検出し，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メソッド呼び出し履歴の変化を複数検出した．その原因は 2 つのバージョンの JHotDraw で同じシナリオを実行したつもりであったが，実行時の動作が変わっていたことにある．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，変化部分と判断されていた．

5.5 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．今回取得したメソッド呼び出し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る舞いの変化を検出できた．実験時には，変化部分を抽出したシーケンス図を主に使って調査したが，シーケンス図から変化部分を人の手で探す必要がないので調査が楽になった．強調表示されているとはいえ，巨大なサイズとなったシーケンス図から変化した部分を探すのはそれなりの労力が必要である．Amida では一画面にはオブジェクトの数はせいぜい 10 個しか表示できず，同様にメソッド呼び出しも 10 個ほどしか表示できない．Amida のシーケンス図の縮小図は 1 つのメソッド呼び出しを，高さのドット数は 1 で，長さのドット数は呼び出し元オブジェクトから呼び出し先オブジェクトまでにあるオブジェクトの数の 2 倍で表しているが，同じオブジェクト内のメソッド呼び出しは縮小図には表示されない．もし，変化部分の抽出機能がなかったら GUI を操作してシーケンス図の表示箇所を変えながら変化部分を探さなければならない．

メソッド呼び出し履歴を人の手で比較する場合，今回の実験で使用したメソッド呼び出し履歴では 5 万回のメソッド呼び出しが行われているため，どこが異なるのかを見つけるだけで困難である．純粋に人の手だけで比較をしようとすると，オブジェクト ID とタイムスタンプといった特徴となる数字は同じ動作をしていてもメソッド呼び出し履歴間で異なるため，比較するときに利用できず，メソッド呼び出し名などの長い文字列を比較して見つけなければならない．2 つのファイルを見比べてもどこに違いがあるのか分からず，変更箇所を

見つけることができなかった．次に適用実験で使用したメソッド呼び出し履歴を diff コマンドで差分検出し，ファイルに出力したが，出力結果はメソッド呼び出し履歴とほぼ同じである 5,434,659byte のファイルになった．diff コマンドの結果を見ると，メソッドの終了記号がほとんど出力されず，差分も数行ごとに細切れに表示されていたため，プログラムの振る舞いを理解することができなかった．また，diff コマンドの出力結果には差分が表示されるが，その差分である部分に対応があるかどうかも分からないため情報を減らしただけでプログラムの変化を見つけることができなかった．さらに，diff コマンドでタイムスタンプとオブジェクト ID を取り除いたメソッド呼び出し履歴で差分検出も行った．このとき diff コマンドの結果を出力したファイルは 1,780,239byte に減ったが，オブジェクト ID やタイムスタンプといった，元のメソッド呼び出し履歴ではどのメソッド呼び出しなのかを特定する情報が抜け落ちていたため，全く役に立たなかった．

本手法は，メソッド呼び出し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る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る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，メソッド呼び出し履歴を取得すれば，入力データの違いによるプログラムの振る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，複数のメソッド呼び出し履歴を取得し，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．

6 関連研究

本研究では，メソッド呼び出し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メソッド呼び出し履歴の比較手法を提案している．メソッド呼び出し履歴の比較手法として，Hoffman ら [6] はメソッド呼び出し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（Longest Common Subsequence，以降 LCS）を求める手法を提案している．LCS 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 [8] は，メソッド呼び出し履歴を，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，大規模なメソッド呼び出し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可能である．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メソッド呼び出し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メソッド呼び出し履歴の比較手法としては，Smit ら [15] 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，その区間単位でメソッド呼び出し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，メソッド呼び出し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，変更部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 SimpleTreePattern-MatchingArgorithm [10] 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，メソッド呼び出し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，メソッド呼び出し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，同じメソッド呼び出し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 [12] 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 [4] 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 [5] を提案している．

7　まとめ

本研究では，木構造の比較に基づくソースコードの変更前後でのメソッド呼び出し履歴の変化の可視化手法を提案した．提案した手法はソースコードの変更によって実行時のメソッドの呼び出し関係が変化することに着目し，同じシナリオを実行して取得したメソッド呼び出し履歴からメソッドの呼び出し関係を取得し，木構造のデータに変換し比較を行うことで，ソースコードの変更によるメソッド呼び出し履歴の変化を検出し，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適用実験により，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，木構造の比較アルゴリズムの改良が挙げられる．メソッド呼び出し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い，入力したメソッド呼び出し履歴が巨大であると比較計算に膨大な時間がかかってしまうという欠点がある．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ができれば，巨大なメソッド呼び出し履歴を比較する場合でも，実用可能となる．また，処理時間の削減に Amida のループ圧縮機能を利用することも考えている．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．この機能をメソッド呼び出し履歴の比較の前に適用し，圧縮によってメソッド呼び出し関係木のノード数を減らすことができれば，比較計算の処理時間を減らすことできると考えられる．ただし，ループ圧縮機能自体の計算時間がかかるため，全体として計算時間を減らすことができるか，どのような場合に効果的な結果を得られるのかを調査する必要がある．

謝辞

本研究において，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究において，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において，逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究において，様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 渡邊 結 氏に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] 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] J. K. Czyz and B. Jayaraman. Declarative and visual debugging in eclipse. *Eclipse Technology Exchange*, 2007.

[4] 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.

[5] 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.

[6] 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.

[7] JHotDraw. `http://sourceforge.net/projects/jhotdraw/`.

[8] 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.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 492–501, 2006.

[10] 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.

[11] 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.

[12] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the 3rd International Workshop on Dynamic Analysis*, pp. 1–6, 2005.

[13] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pp. 221–230, May 2001.

[14] 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.

[15] 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.

[16] Unified Modeling Language(UML)1.5 specification, March 2003. OMG.

[17] 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.

[18] N. Wilde and R. Huitt. Maintenance support object-oriented programs. *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038–1044, 1992.

[19] 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.