特別研究報告

題目

メソッドの実行経路の分類を用いた実行履歴可視化ツール

指導教員
井上 克郎 教授

報告者
松村 俊徳

平成 26 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

メソッドの実行経路の分類を用いた実行履歴可視化ツール

松村 俊徳

内容梗概

ソフトウェア開発において開発全体の 5 割から 8 割が保守工程に費やされる．保守工程では，様々な目的でソフトェアに対して変更が適用されるが，変更を行うための土台としてプログラムの理解が必要となる．プログラム理解のための基礎となる情報源はソースコードや設計書などの静的なものであるが，プログラムの実行時の振る舞いを調べることはプログラムの理解の大きな助けとなり，実際に多くの開発者がプログラムの理解に実行時の動的な情報を利用している．

実行時情報を閲覧する手法として Omniscient Debugging が挙げられる．Omniscient Debugging ではプログラムの実行履歴を記録しておき，実行後に記録した履歴からプログラムの任意の時点の実行時情報を取得できるため，実行を遡り調査を行うといったことが可能となる．しかし，ここで問題となってくるのは，Omniscient Debugging で一度に閲覧可能なのはある 1 つの時点での実行時情報のみであるため，調べる対象となるプログラム文へ複数回実行が到達している場合，開発者は各到達に関して個別に観測しなければならないということである．

本研究ではプログラムの意味的まとまりであるメソッドに焦点を当て，メソッドの複数回の実行の実行履歴を一覧として可視化する手法を提案し，ツールを実装した．メソッドの実行履歴の一覧を可視化するにあたり，単純に羅列し可視化しただけでは実行履歴の量が膨大となり，開発者にとって閲覧する労力が大きくなってしまうことからメソッドの実行経路に着目し実行履歴の分類を行い，提示をする方式を実現した．

また，本研究では，ツールの適用可能性の調査のため，分類による効果と実行履歴の取得にかかる時間という側面から調査を行い，多くのメソッドに対して適用可能であることを確認し，実際にツールを適用した場合に実行履歴の一覧の可視化が行えていることを確認した．

主な用語

動的解析
プログラム理解

実行履歴

目 次

1　まえがき

ソフトウェア開発において開発全体の 5 割から 8 割が保守工程に費やされる [1]．保守工程とは運用中のソフトウェアを正常に稼働させ続けるための工程であり，ソフトウェアに対し多くの変更が適用される．その変更の目的にはバグの修正や，新たな機能の追加，パフォーマンスの改善，環境の変化に伴った修正などが挙げられる．

保守工程における重要なプロセスとしてプログラム理解がある．このプログラム理解が保守工程の半分以上を占めるということが知られている [2]．保守工程では，様々な目的でソフトェアに対して変更が適用されるが，変更を行うにはプログラムの理解が必須である．プログラム理解では，多くの開発者がプログラム全体の理解を行うのではなく，適用したい変更に関係する部分のプログラム理解を行い，意図した変更を正しく行うためにはどうしたらよいかということや，他の機能へどのような影響があるかというようなことを調査する．

プログラム理解のための基礎となる情報源はソースコードや設計書などの静的なものであるが，プログラムの実行時の振る舞いを調べることはプログラムの理解の大きな助けとなり，実際に多くの開発者がプログラムの理解に実行時の動的な情報を利用している [3, 4]．実行時情報を閲覧する方法として GDB[5] などのデバッガが挙げられる．GDB などのデバッガでは，ブレークポイントの設定を行ってプログラムを一時停止し，あるプログラム文に実行が到達した時点の実行時情報の閲覧が可能である．しかし実行の巻き戻しはできないため，例えば，あるプログラム文へ到達した原因を実行を遡って調査するといったことは不可能である．

このような問題に対処可能な実行時情報を閲覧する手法の 1 つとして Omniscient Debugging[6] が存在する．Omniscient Debugging ではプログラムの実行履歴を記録しておき，実行後に記録した履歴からプログラムの任意の時点の実行時情報を取得できるため，実行を遡り調査を行うといったことが可能となる．しかし，ここで問題となってくるのは，Omniscient Debugging では一度に閲覧可能なのはある 1 つの時点での実行時情報のみであるため，調べる対象となるプログラム文へ複数回到達している場合，開発者は各到達に関して個別に観測しなければならないということである．

本研究ではプログラムの意味的まとまりであるメソッドに焦点を当て，複数回呼び出されるメソッドを対象とし，メソッドの実行時情報の閲覧を支援する目的で，メソッドの実行履歴の一覧を可視化する手法を提案し，ツールを実装した．メソッドの実行履歴の一覧を可視化するにあたり，単純に羅列し可視化しただけでは実行履歴の量が膨大となり，開発者にとって閲覧する労力が大きくなってしまうことからメソッドの実行経路に着目し実行履歴の分類を行い，提示をする方式を実現した．

また，本研究では，ツールの適用可能性の調査のため，分類による効果と実行履歴の取得

にかかる時間という側面から調査を行い，多くの適用可能であることを確認し，実際にツールを適用した場合に実行履歴の一覧の可視化が行えていることを確認した．

以降，2 章では，本研究の背景について述べる．3 章では，提案する手法について述べる．4 章では，ツールの実装について述べる．5 章では，ツールの適用実験について述べる．6 章では，関連研究について述べる．7 章では，本研究のまとめと今後の課題を述べる．

2　背景

この節では，背景としてプログラムの実行時情報の閲覧方法を述べ，従来の閲覧方法における問題点と，そのような問題点を解決する Omniscient Debugging について述べる．

2.1　実行時情報の閲覧

プログラムのテストやデバッグにおいて，実行時情報が利用される．実行時情報とはプログラムを実行した際のある時点での情報であり，その情報には様々なものが含まれる．例えば変数の状態やスタックトレースなどである．開発者にとって必要な実行時情報は状況に応じて異なり，それぞれの状況に適した情報を含む実行時情報が利用される．例えば，テストの網羅率を評価するためにカバレッジを計算する場合には，実行時情報としてプログラム内の行番号を含むものが利用される．

実行時情報を閲覧するには大きくわけて 2 つの方法がある．1 つはプログラムを実行しながら実行時情報を記録し，プログラムの実行が終わった後に記録した情報を閲覧する方法であり，もう 1 つはプログラムを実行中に実行を一時停止し，実行時情報を閲覧する方法である．前者の方法の代表的な例が printf デバッグであり，後者の代表的な例がブレークポイントデバッグである．

プログラムの実行時情報を閲覧するために，printf デバッグやブレークポイントデバッグでは実行時情報を見たい地点や見たい情報を，開発者が予め指定しておかなければならないという問題がある．実行時情報を見たい地点や情報が予めわかっている場合はこれらの方法に問題はないが，デバッグにおいて一般的に欠陥の箇所を特定することは困難であるように，常に適切な地点を指定できるとは限らない．結果として，開発者は実行時情報を見たい地点を変更しながら何度もプログラムを実行するか，もしくは複数の地点を指定し多くの実行時情報から欠陥に関係する情報を探すこととなる．

2.2　Omniscient Debugging

Omniscient Debugging はデバッグ手法の 1 つであり，プログラム実行の完全な実行履歴を記録することで，任意の時点の実行時情報を閲覧することができるという特徴を持つ．この特徴をうまく活用できれば，障害が起きた時点から実行を遡り欠陥を特定することも可能となる．

Omniscient Debugging は主に Capture と Replay という 2 つのステップから構成される．Capture ではプログラムの実行履歴の記録を行い，Replay では，記録した情報を基に実行時情報を再現する．記録する情報の詳細さと再現にかかる手間にはトレードオフの関係があり，より詳細な実行履歴を記録すれば，再現の段階で行う処理が減るが，記録する情報の量

やオーバーヘッドが増加する．逆に再現に必要な最低限の情報のみを記録すれば，再現で行う処理が増加する．

プログラムの実行を遡ることができるため，実行履歴の記録によるオーバーヘッドを無視すれば，Omniscient Debugging は，実行時情報の閲覧に有用である．しかし，Omniscient Debugging では同時に閲覧可能な時点は 1 つであるため，複数回到達しているプログラム文の地点に関して実行時情報を閲覧したいような場合には，それぞれの到達について繰り返し閲覧しなければならない．

3　提案手法

本研究では，メソッドの実行経路の分類を用いた実行履歴の可視化手法を提案する．提案手法は次に示す 4 つのステップから構成される．提案手法では，事前に対象となるメソッドが呼び出される実行を記録しておく必要があり，開発者はプログラムの実行後にメソッドを指定し実行履歴を閲覧する．

1. プログラムの実行ログの記録
2. 実行ログに基づくメソッドの実行履歴の再現
3. 実行履歴の実行経路に基づく分類
4. 実行履歴の可視化

以降，各ステップについての詳細を説明する．

3.1　プログラムの実行ログの記録

Java プログラムの実行をメソッド単位で再現するために，1 つのメソッドに注目したとき，そのメソッドの外部とのすべてのデータのやりとりを実行ログとして記録する．実行ログはイベントの列として表現し，各イベントには，それぞれのイベントの種類に応じて対応する値を記録する．

Java プログラムにおけるメソッドの外部とのやりとりを表 1 に示す．例えば，ReturnValue イベントというメソッドの返り値に関するイベントでは，返り値を記録し，Call イベントというメソッド呼び出しに関するイベントでは，引数の値を記録する．例として，図 1 に示す Java プログラムの実行により記録されるイベントの列は表 2 のようになる．表中の ID1,ID2,ID3 はそれぞれ，String 配列型オジェクトの ID，PrintStream 型オブジェクトの ID，Test 型オジェクトの ID，を表している．

Java プログラムの実行ログの記録には Java プログラムのバイトコードに実行ログを書き出す命令を埋め込み，命令を埋め込んだプログラムを実行し実行ログを記録する．実行ログの記録ではプログラム全体に命令を埋め込み，実行開始から終了までのプログラム全体の実行ログを記録する．命令を埋め込む部分を分析対象となるメソッドのみに限定することも可能であり，限定した場合は実行ログの記録量や記録時間の削減などのメリットがあるが，その場合他のメソッドを追加で分析するには，再びプログラムの変換・実行を行わなければならなくなるため，本研究で作成したツールではプログラム全体の実行ログを記録する方式を採用する．

表 1: イベントの種類

イベント名	イベントの意味	値
Entry	メソッドの開始	引数の値
NormalExit	メソッドの正常な終了	-
ExceptionalExit	メソッドの例外発生による終了	-
Call	メソッドの呼び出し	引数の値
ReturnValue	呼び出したメソッドの返り値の受け取り	返り値
GerField	フィールドの読み出し	読み出した値
ObjectCreation	オブジェクトの生成	オブジェクト ID
NewArray	配列の生成	オブジェクト ID
MultiNewArray	多次元配列の生成	オブジェクト ID
ArrayLoad	配列の値の読み出し	読みだした値
ArrayLength	配列の長さの参照	配列の長さ
Throw	throw 文による例外の送出	例外オブジェクト ID
Catch	Catch ブロックによる例外の補足	補足したオブジェクト ID
InstanceOf	instanceof 命令の実行	Boolean 値

```
1.  package test;
2.  public class Test {
3.
4.      public static void main(String[] args) {
5.          Test obj = new Test();
6.          System.out.println(obj.square(10));
7.      }
8.
9.      public int square(int x){
10.         int y = x*x;
11.         return y;
12.     }
13. }
```

図 1: サンプルコード

3.2 実行ログに基づくメソッドの実行履歴の再現

実行履歴の再現では，開発者が指定したメソッドに対して，そのバイトコードの命令を実行ログの情報を用いて実行しなおすことで以下の実行ログに記録されていない情報を復元する．

- バイトコード命令の実行順序
- 各命令を実行した時点での，ローカル変数の状態
- 各命令を実行した時点での，Java 仮想マシンのオペランドスタックの状態

再現の方法としては，まず実行ログからイベントを読み出す．そして，読み出したイベントに対応するバイトコード命令番号までの各バイトコード命令を適用し，ローカル変数の状態，オペランドスタックの状態を更新していく．各バイトコード命令の適用の結果は，順次実行履歴に追加する．対応するバイトコード命令番号まで各バイトコード命令を適用し終わるれば，次のイベントを読み出す．以上のことをメソッドの終了イベントに達するまで繰り返すことで実行履歴を生成する．なお，1 つのバイトコード命令を処理するたびにその結果を実行履歴に追加していく．

例として図 1 のプログラムの square メソッドのバイトコードを図 2 に，再現された実行履歴を表 3 に示す．表 3 において，ローカル変数は，変数名，型名，値の 3 つ組の集合として，オペランドスタックは型名と値の組のスタックとして表現している．バイトコード命令に対応するソースコードの行番号はバイトコードのデバッグ情報から復元している．

表 3 から分かるように，この square メソッドのソースコード上における実行経路は，メソッド開始→ 10 行目→ 11 行目となっており，ソースコード上の 10 行目まで実行した時点でのローカル変数 y の値は 100 となっている．

対象のメソッドが n 回実行されていた場合，それぞれの実行に対応した n 個の実行履歴を生成し，続くステップである実行履歴の分類や実行履歴の可視化に使用する．

なお，実行履歴を再現するのは，対象としているメソッドの実行のみであり，その他のメソッドについては再現を行わない．ゆえに実行履歴の再現に利用する実行ログは対象としているメソッドの開始から終了までの部分的イベント列となる. 例えば，図 1 のプログラムの square メソッドに関して実行履歴の再現を行う場合には表 2 におけるイベント列のうちイベント番号 6,7 のイベントのみを利用する．

3.3 実行履歴の実行経路に基づく分類

実行履歴の再現により得た実行履歴が膨大であった場合，一覧を提示しただけでは開発者が興味のある実行履歴を探す労力が大きくなると予想される．そこで，効率よく閲覧するた

表 2: 図 1 のプログラムの実行により記録されるイベント列

イベント番号	イベント名	値	イベントの説明
0	Entry	ID1	main メソッドの開始
1	Call	-	Test クラスのコンストラクタの呼び出し
2	Entry	-	Test クラスのコンストラクタの開始
3	Exit	-	Test クラスのコンストラクタの終了
4	GetField	ID3	System クラスの out フィールドの読み出し
5	Call	ID2,10	square メソッドの呼び出し
6	Entry	ID2,10	square メソッドの開始
7	Exit	-	square メソッドの終了
8	ReturnValue	100	square メソッドの返り値
9	Call	ID3,100	println メソッドの呼び出し
10	ReturnValue	void	println メソッドの返り値
11	Exit	-	main メソッドの終了

表 3: 再現された実行履歴

バイトコード命令番号	ソースコード行番号	ローカル変数	オペランドスタック
0	0	(this,test/Test,ID2)(x,int,10)	
2	10	(this,test/Test,ID2)(x,int,10)	(int,10)
3	10	(this,test/Test,ID2)(x,int,10)	(int,10)(int,10)
4	10	(this,test/Test,ID2)(x,int,10)	(int,100)
5	10	(this,test/Test,ID2)(x,int,10)(y,int,100)	
8	11	(this,test/Test,ID2)(x,int,10)(y,int,100)	(int,100)

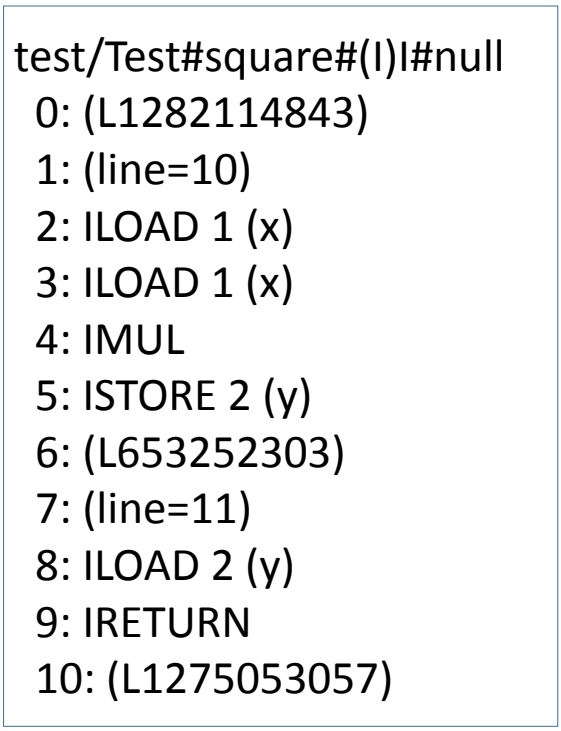

図 2: 図 1 における square メソッドのバイトコード

めに実行履歴を実行経路に基いて分類する．実行履歴にはバイトコード命令の実行順序の情報が含まれているので，それを利用して実行経路を判断する．分類には次の 3 つの分類基準を用いる．これらは実行履歴を等しいとみなす基準である．

Line 実行したバイトコード命令番号の集合が等しければ同じ分類

Frequency 1 回だけ実行したバイトコード命令番号の集合と，2 回以上実行したバイトコード命令番号の集合が等しければ同じ分類

Path バイトコード命令の実行順序が等しければ同じ分類

Line による分類を行うことで，命令の実行の有無を区別できるため，例えば if 文内を実行している実行履歴に興味がある場合には，そのような実行履歴にのみ注目できる．Frequency による分類では Line による分類では区別できない複数回の実行の有無を区別できる．したがってループ内を繰り返し実行している実行履歴に興味がある場合などに有効である．Path による分類では，命令の実行順序で分類を区別しているため，Line や Frequency よりも細かな条件を満たす実行履歴に着目することが可能となる．

図 3 のプログラムを例にとり，分類の具体例を示す．図 3 のプログラムの loop メソッドに関して実行履歴を再現したとする．この loop メソッドは引数の int 値により実行経路が変化するメソッドであり，main メソッドから計 10 回，引数の値を 0 から 9 まで変化させながら呼び出されている．図 4 は loop メソッドの実行経路を示したものであり，メソッド実行中に通過した行番号の列として表現している．

Line による分類結果が図 5 である．loop(0),loop(4),loop(8) では 12 行目を通過していないのに対し，その他の実行履歴では 12 行目を通過しているためこのような結果となっている．

図 6 は Frequency による分類結果である．1 つ目の分類は 12 行目を通過していない，2 つ目の分類は 1 回のみ 12 行目を通過，3 つ目の分類は複数回 12 行目を通過している分類となっている．

図 7 は Path による分類結果であり，実行経路が等しい実行履歴が同じ分類に含まれている．

なお，先に示した例からも分かるように，分類の粒度は Line が最も粗く，Path が最も細かい分類となる．これら 3 つの分類を目的に合わせて開発者が選択できるようにすることで，膨大な量の実行履歴の閲覧をせずにすむ．

3.4 実行履歴の可視化

実行履歴をユーザーが閲覧できるように可視化を行う．実行履歴からは，任意の時点でのローカル変数の情報，実行経路を取得することが可能である．分類を行った実行履歴を可視

```
1.  public class Test {
2.      public static void main(String[] args) {
3.              for(int i=0;i<10;i++){
4.                      loop(i);
5.              }
6.      }
7.
8.      public static void loop(int i){
9.              int j = i % 4;
10.
11.             for(int k = 0;k < j;k++){
12.                     System.out.println("i="+i+",j="+j+",k="+k);
13.             }
14.     }
15. }
```

図 3: 分類を示すためのサンプルプログラム

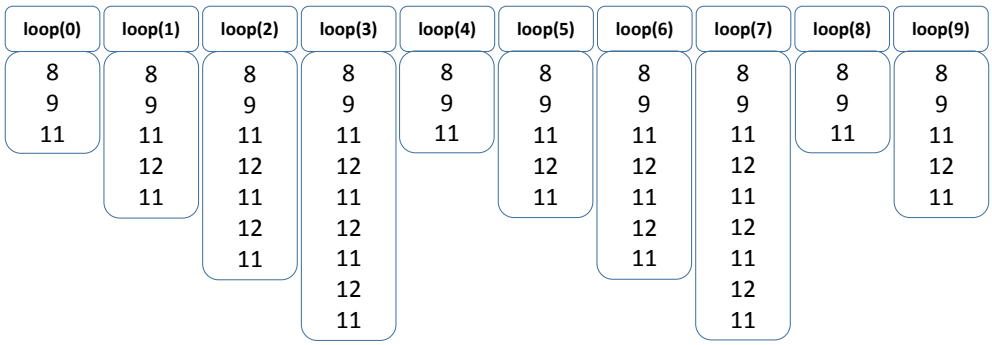

図 4: loop メソッドの実行経路

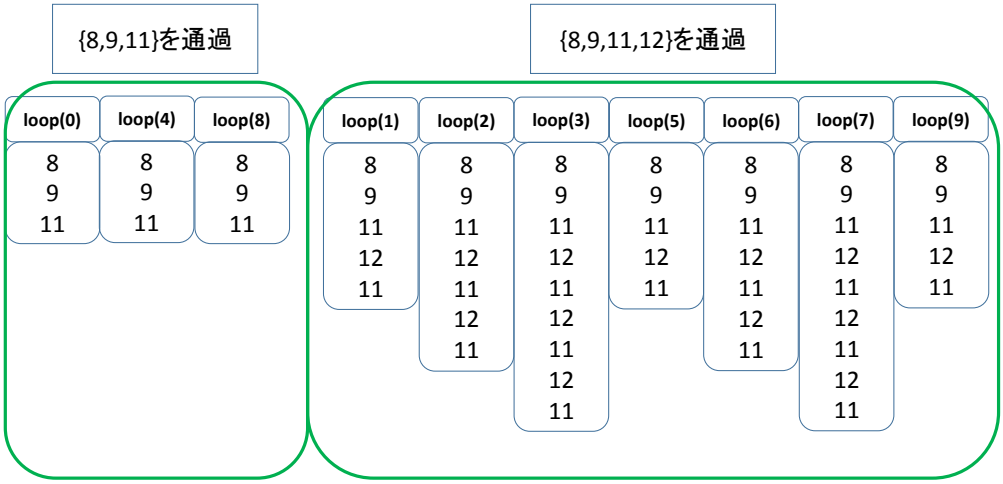

図 5: Line による分類

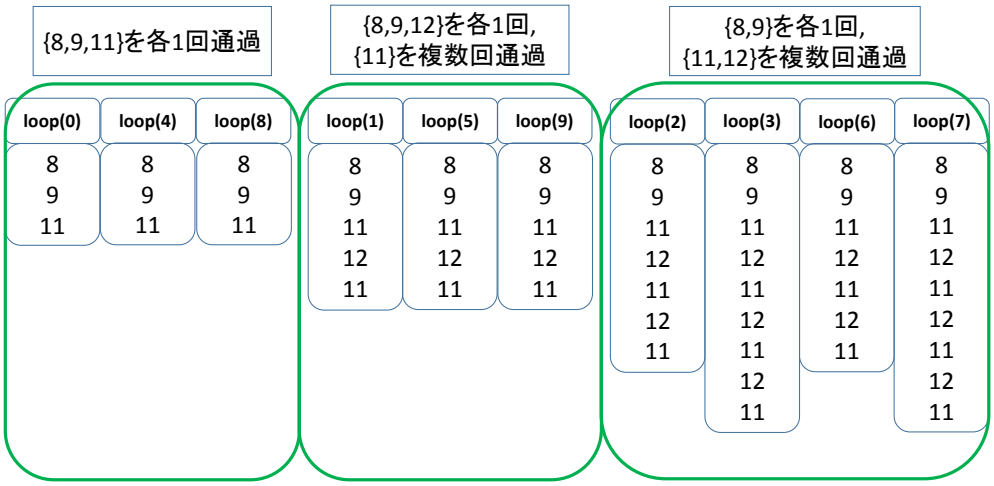

図 6: Frequency による分類

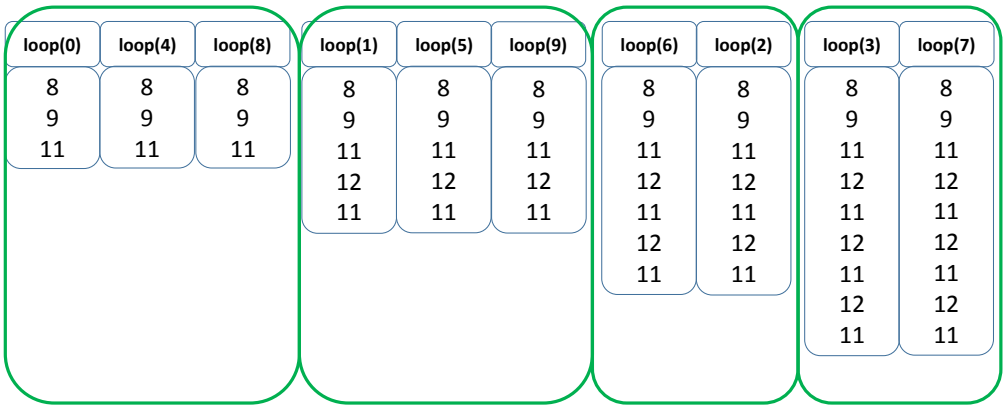

図 7: Path による分類

化する際には，各分類から実行履歴をひとつずつ選び出し可視化を行う．可視化する情報は次のものである．

- ローカル変数の状態
- 実行経路

そして，実行履歴からメソッドの挙動がすべて分かるので，指定した行に関して，その行を通過する時のローカル変数の状態の一覧についても可視化を行う．この情報を表示することで，ある行を通過するすべての時点においての変数の情報を閲覧可能となる．

図 8: ツール起動直後の画面

4　ツールの実装

本節では，Java を用いて実装した実行履歴可視化ツールの機能及び使用方法について説明する．本ツールにより，指定したメソッドの実行履歴の一覧を分類基準を変更しながら閲覧することができる．

本ツールの入力には，対象とするプログラム，プログラムの実行ログ，クラス名，メソッド名を指定する．以下にツール起動に関する手順を示す．

1. 対象となるプログラムに実行ログを書き出すための変換を行う．
2. 変換後のプログラムを実行し，実行ログを記録する．
3. プログラム，プログラムの実行ログ，クラス名，メソッド名を指定しツールを起動する．

図 8 は本ツールを起動した直後の画面であり，入力には図 3 のプログラムの loop メソッドを指定している．初期状態では，各実行履歴は分類されずに横方向に並んで表示される．画面上部には，各種制御ボタンと分類基準変更用のコンボボックスがある．

以降，制御ボタン，分類基準変更機能，実行履歴の表示内容について説明する．

表 4: 制御ボタンの説明

ボタン名	説明
StepBack	選択している実行履歴のポインタを１つ戻す
StepNext	選択している実行履歴のポインタを１つ進める
StepBackAll	すべての実行履歴のポインタを１つ戻す
StepNextAll	すべての実行履歴のポインタを１つ進める
EntryPoint	選択している実行履歴のポインタをメソッド開始位置に設定
ExitPoint	選択している実行履歴のポインタをメソッド終了位置に設定
EntryPointAll	すべての実行履歴のポインタをメソッド開始位置に設定
ExitPointAll	すべての実行履歴のポインタをメソッド終了位置に設定

4.1 制御ボタン

各実行履歴には，現在時点を示すポインタがあり，そのポインタの時点でのローカル変数の状態が表示される．制御ボタンはそのポインタを制御するためのボタンである．

表 4 に各制御ボタンの説明を示す.

4.2 分類基準の変更機能

コンボボックスには，“None”，“Line”，“Frequency”，“Path” の４つの選択肢があり，実行履歴の分類基準を変更できる．“None” を選択すると分類を行わずにすべての実行履歴が表示される．

実行履歴の分類を行った場合には，それぞれの分類から代表となる 1 つの実行履歴が表示される．図 9, 図 10, 図 11 はそれぞれ Line，Frequency，Path による分類を行った状態の画面である．

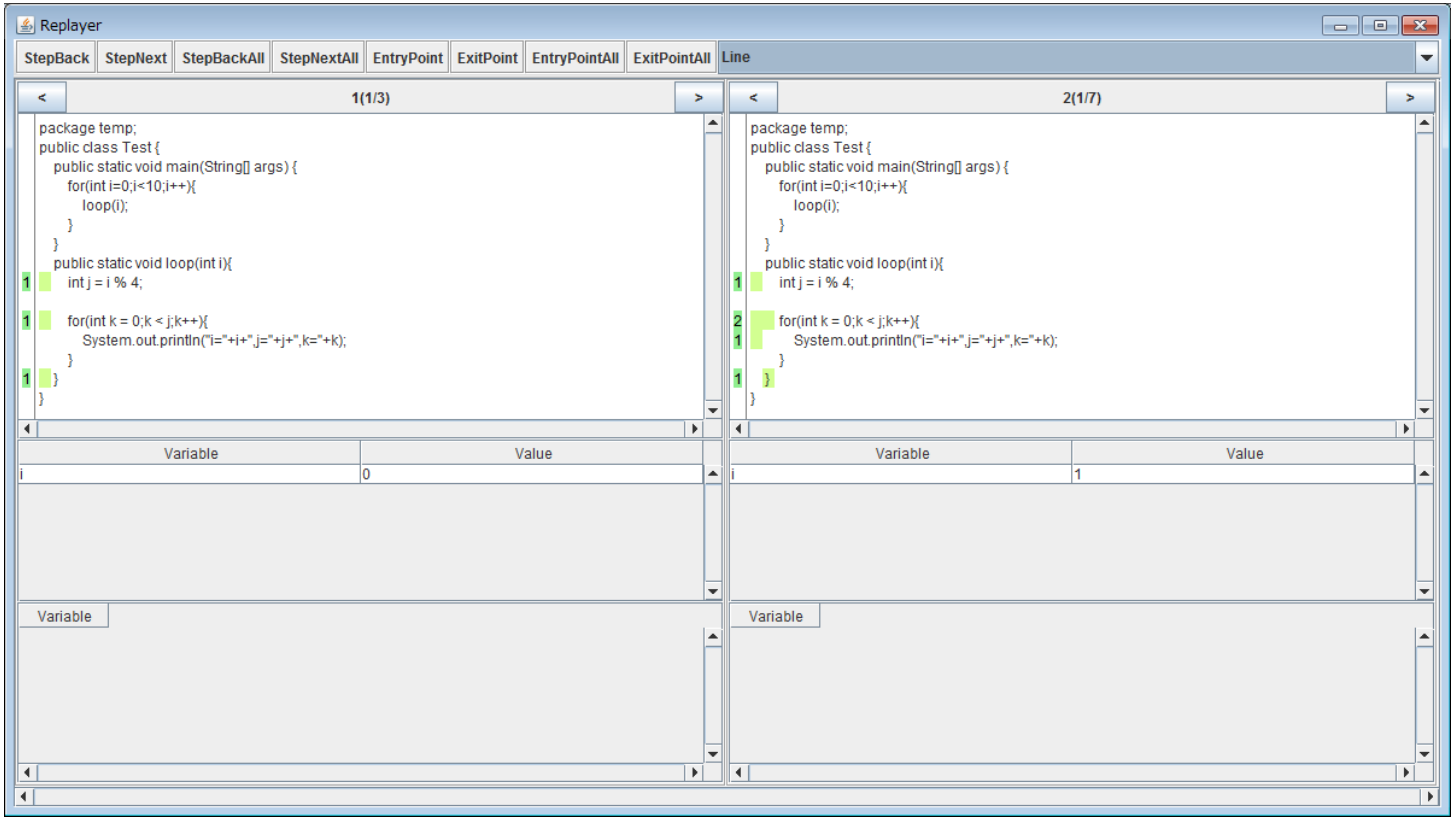

図 9: Line による分類

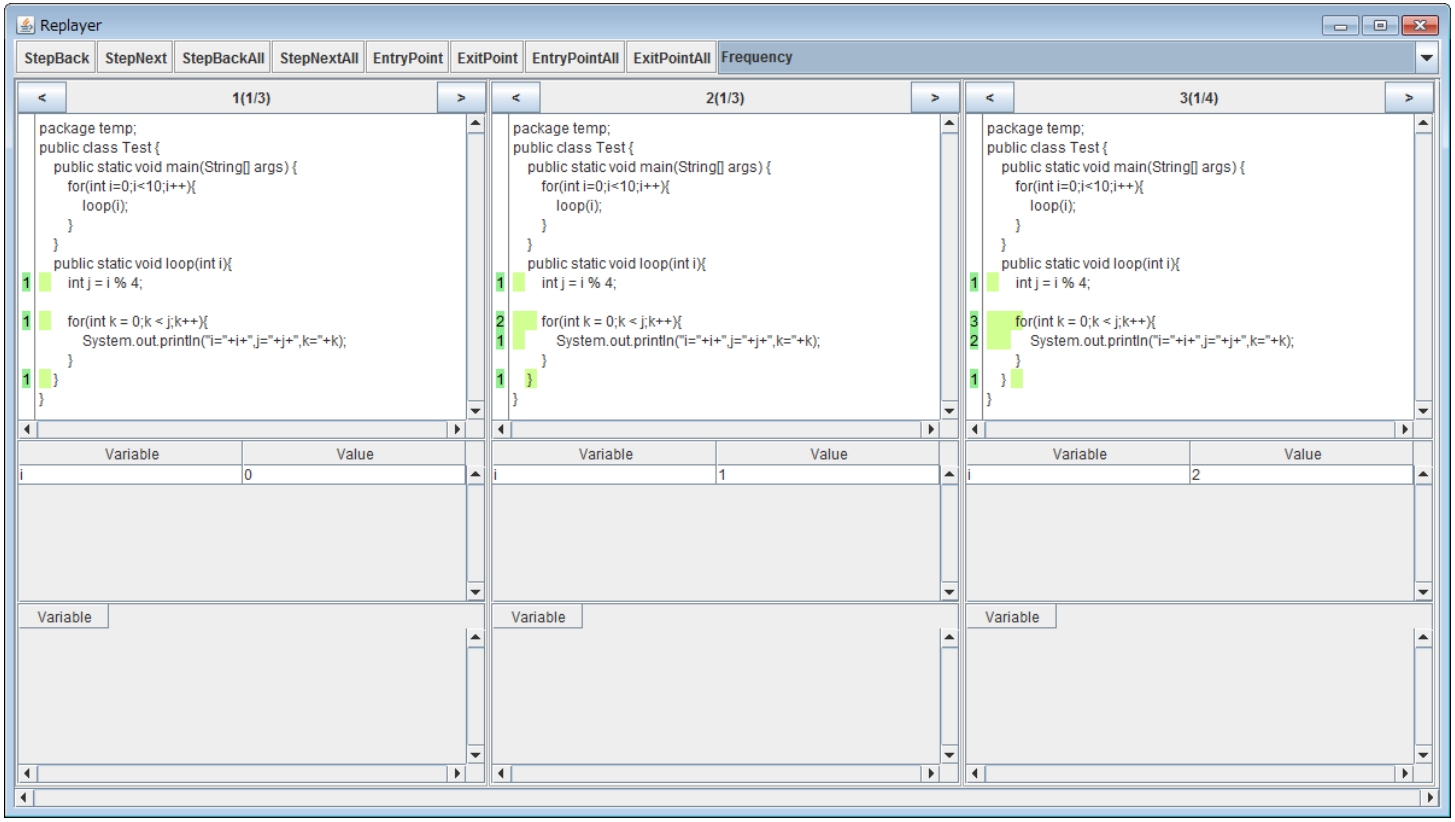

図 10: Frequency による分類

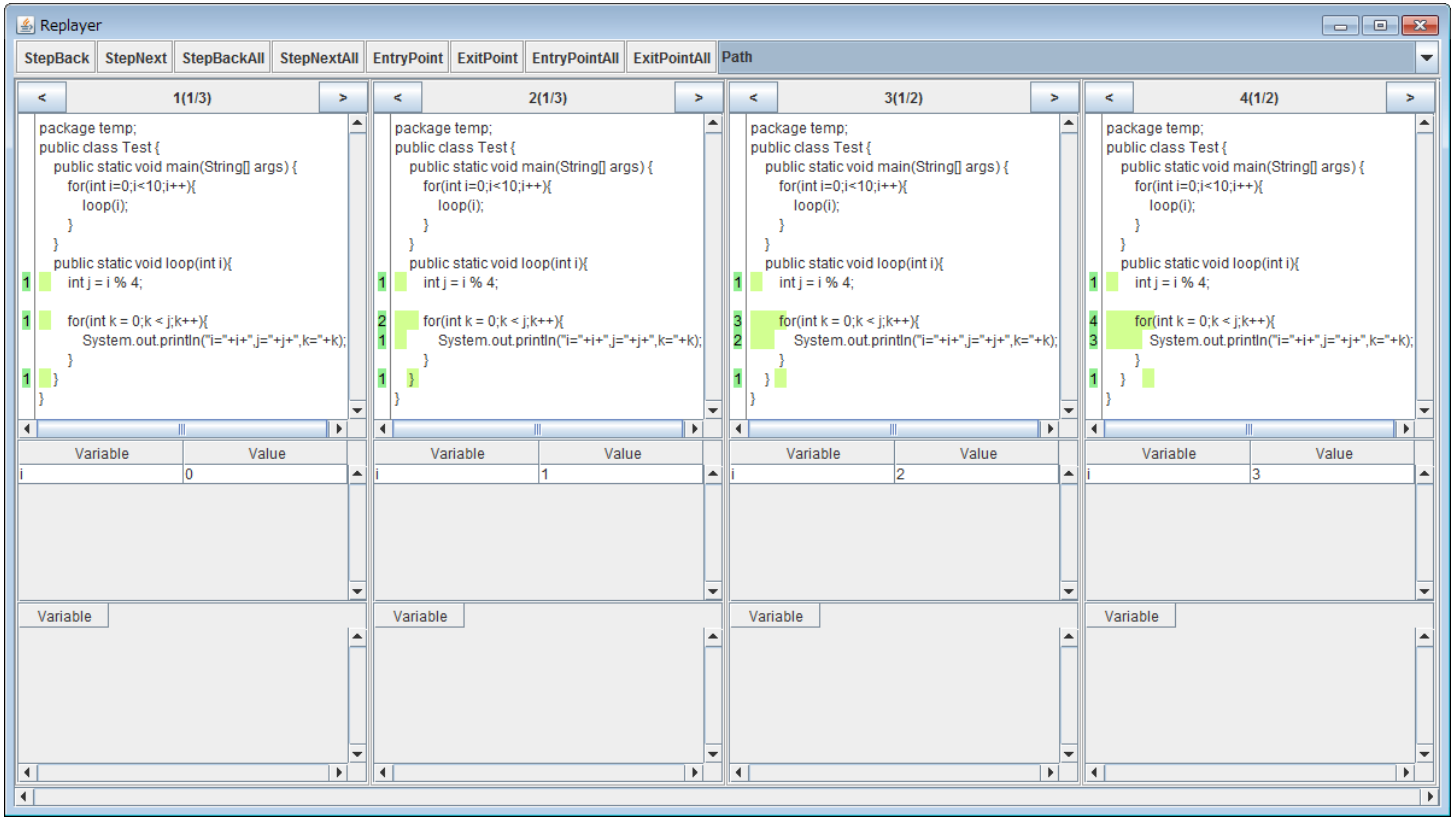

図 11: Path による分類

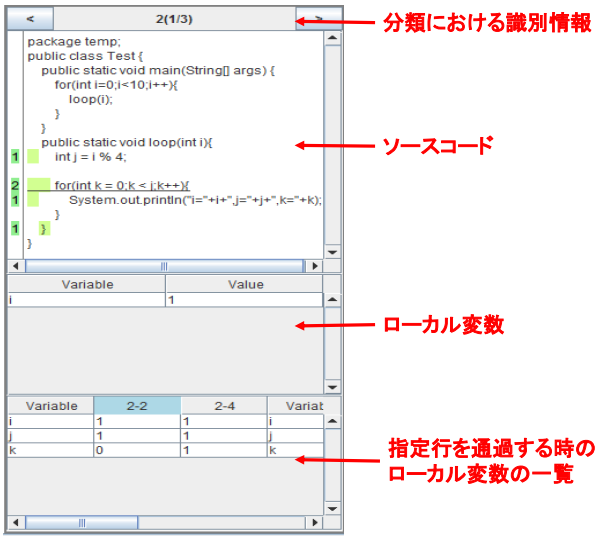

図 12: 実行履歴の表示内容

4.3 実行履歴の表示内容

図 12 は，図 8 における 1 つの実行履歴を拡大したものである．図 12 中に示しているように，1 つの実行履歴が表示している内容は大きく次の 4 つに分かれる．

- 分類における識別情報
- ソースコード
- ローカル変数
- 指定行を通過する時のローカル変数の一覧

分類における識別情報では，何回目のメソッド呼び出しか，分類内におけるインデックス，分類中の実行履歴の数が表示されている．図 12 では “2(1/3)” となっており，これは，2 回目のメソッド呼び出しであり，分類内のインデックスは 1, 分類中の実行履歴の数は 3 ということを表している．また，識別情報の左右にはボタンがついており，押すことで分類内の別の実行履歴を表示させることができる．

ソースコードの部分には，対象としているメソッドのソースコード，通過した行，通過した回数，実行経路の情報が表示されている．通過した行はガター部分の背景色が色付けされ，通過した回数はガター上に数字が描かれている．そして，実行経路は，ソースコードの本体部分の背景色を色付けすることで表現している．ソースコード部分を格子状に分割し，

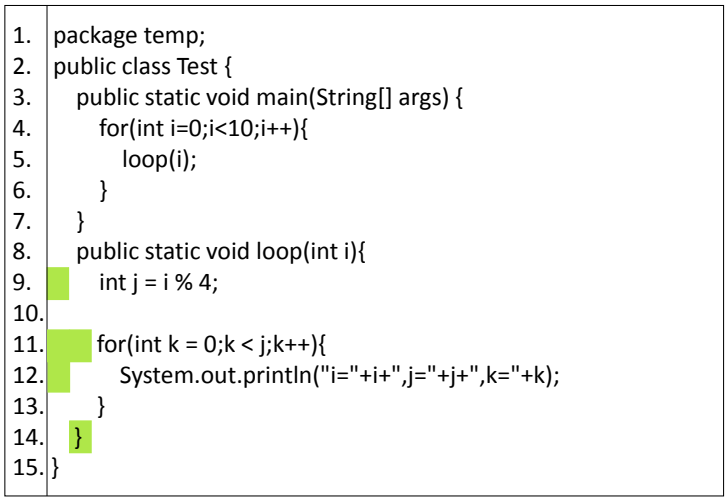

図 13: 実行経路の表示例

通過した行の背景を色付けしていく．プログラムの実行が，行番号が増加する方向 (すなわちソースコード上では下方向) に進んでいく間は，同一の列の背景を色付けし続ける．もし，繰り返し文の実行などで，行番号が減少する方向 (すなわちソースコード上では上方向) にプログラムが進行する場合には色付けする列を 1 つ右側の列にずらす．図 13 に示した例の実行経路は，$9 \to 11 \to 12 \to 11 \to 14$ となる．

ローカル変数の部分には，実行履歴の現在時刻における変数の状態の一覧が表示されている．現在時刻は “StepNext” や “StepBack” 等のボタンで操作することができる．

指定行を通過する時のローカル変数の一覧部分には，ソースコード中の行を指定すると，その行を通過する全ての時刻においての変数の状態の一覧が表示される．また，時刻を指定すると現在時刻を指定した時刻に設定できる．

5 実験

本研究では，提案手法である実行経路による分類により，実行履歴がどの程度分類できるのか, そして実行の再現にはどの程度時間がかかるのかを調査するために実験を行った.

調査のための実験には DaCapo ベンチマーク [7] 内に含まれるアプリケーションを使用して，表 5 に示す環境で実行経路による分類数の計算および再現にかかる時間の計測を行った. DaCapo ベンチマークはアプリケーションの実行規模をオプションで選択できるが，本実験では default とした．実験に使用した Dacapo ベンチマーク内のアプリケーションは，batik と fop であり，表 6 に batik と fop について，それぞれのアプリケーションの実行時間と，実行ログを書き出すための変換を行った後のアプリケーションの実行時間，そして，実行ログから全メソッドの開始位置を収集する前処理にかかる時間を示す.

5.1 実行履歴の分類調査

batik, fop 内の各メソッドに関して実行を再現し，分類を行った結果について説明を行う. 対象となるメソッド数は batik では 3131，fop では 3611 である．図 14 に batik,fop におけるメソッド実行回数の分布を示す．図 14 において縦軸は実行回数を表し，横軸はメソッドを実行回数で昇順に並べている.

図 15,16,17 はそれぞれ Line,Frequency,Path による分類結果をグラフで示したものである．横軸は分類数を，縦軸は全体のメソッドに対する割合を累積度で示している．図 15 から，全体のメソッドのうち 90%以上が Line による分類で分類数が 2 以下になることが読み取れる．Frequency と Path についても同様の結果となっている.

図 15,16,17 は，実行を再現したすべてのメソッドについてのデータであるが，ループ・分岐の無いメソッドや，呼び出し回数が 1 回のメソッドでは実行経路は 1 パターンであり，そのようなメソッドに対しては分類は効果を発揮しない．そこで，ループもしくは分岐を含み, 呼び出し回数が 2 回以上であるメソッド (batik では 903，fop では 867 個のメソッド) を対象とした結果が図 18,19,20 である．図より，Line, Frequency では 3 通り以下，Path では 5 通り以下となるメソッドの割合がおよそ 90%であることから，多くのメソッドに対して実行経路による分類に効果があることが確認できる.

表 5: 実験環境

OS	Microsoft Windows8.1
RAM	256GB
CPU	Intel Xeon 2.90GHz

表 6: アプリケーションの実行時間，前処理の実行時間

	batik	fop
変換前	4.44[s]	2.80[s]]
変換後	33.67[s]	36.914[s]
前処理	651[m]	1045[m]

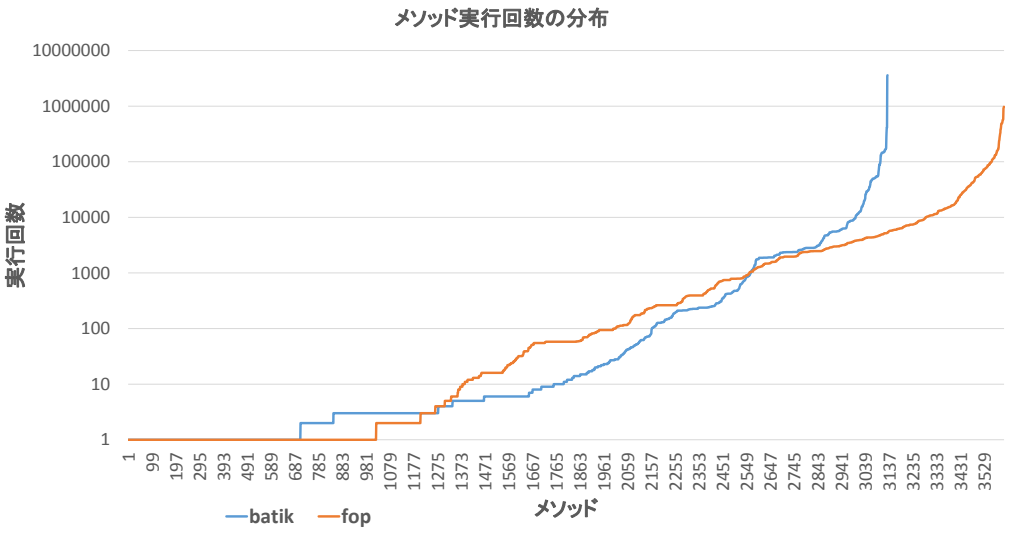

図 14: メソッド実行回数の分布

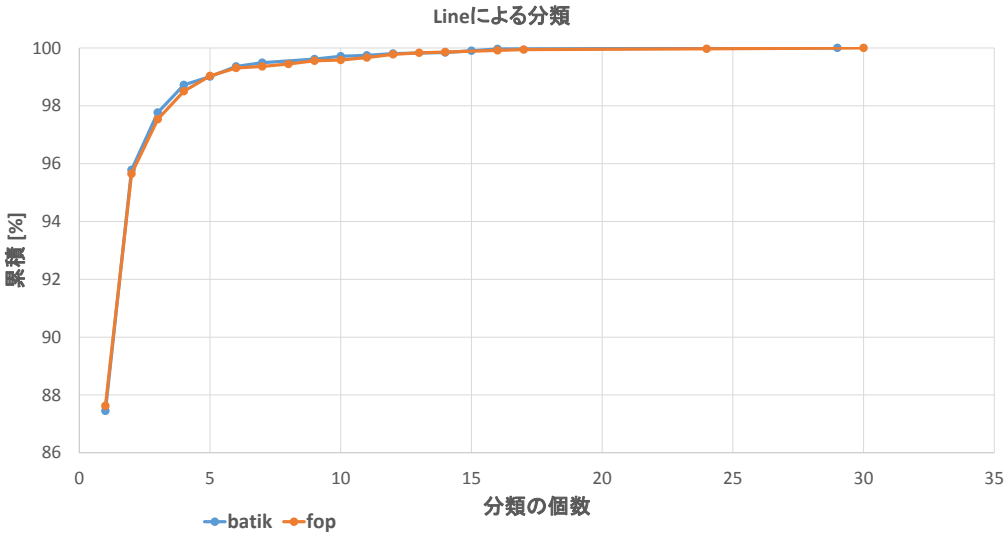

図 15: すべてのメソッドに対する Line による分類

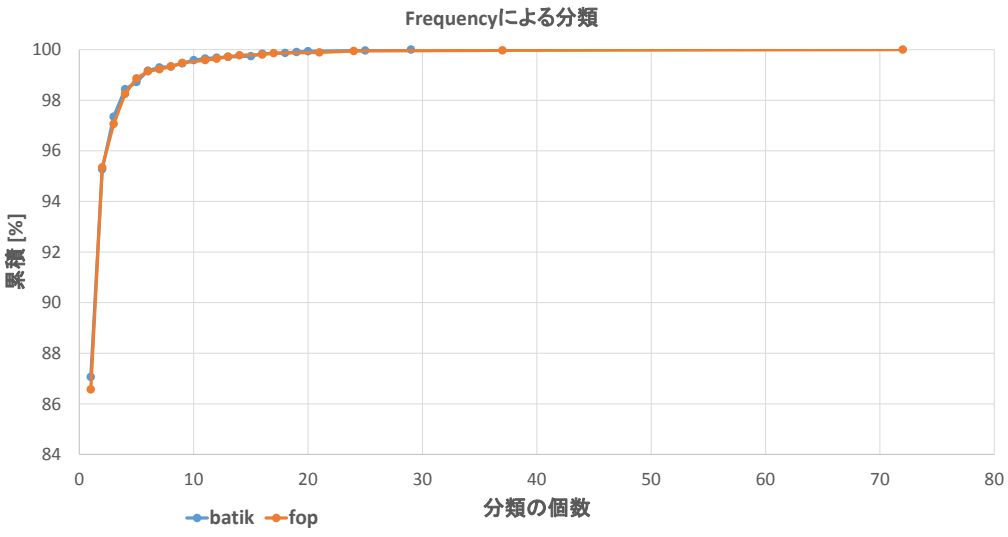

図 16: すべてのメソッドに対する Frequency による分類

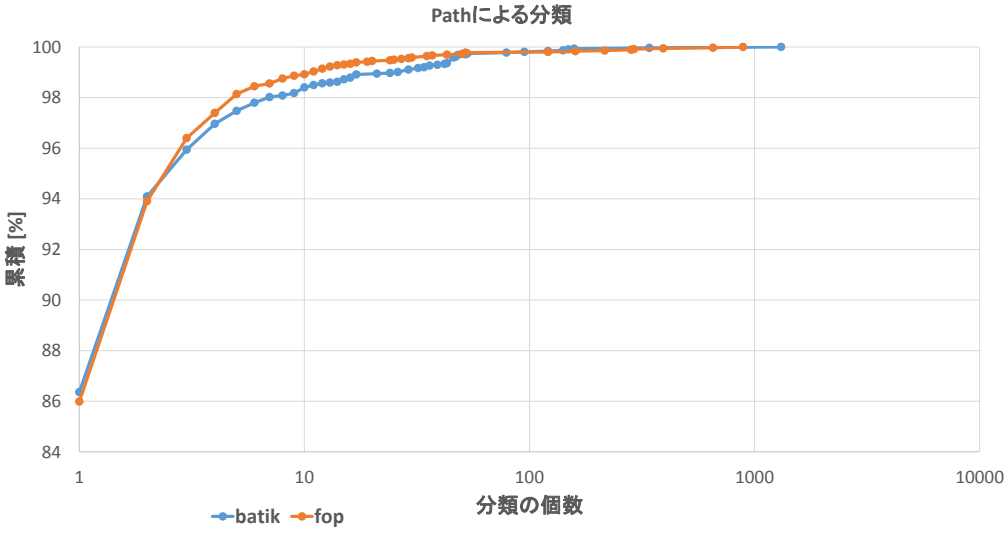

図 17: すべてのメソッドに対する Path による分類

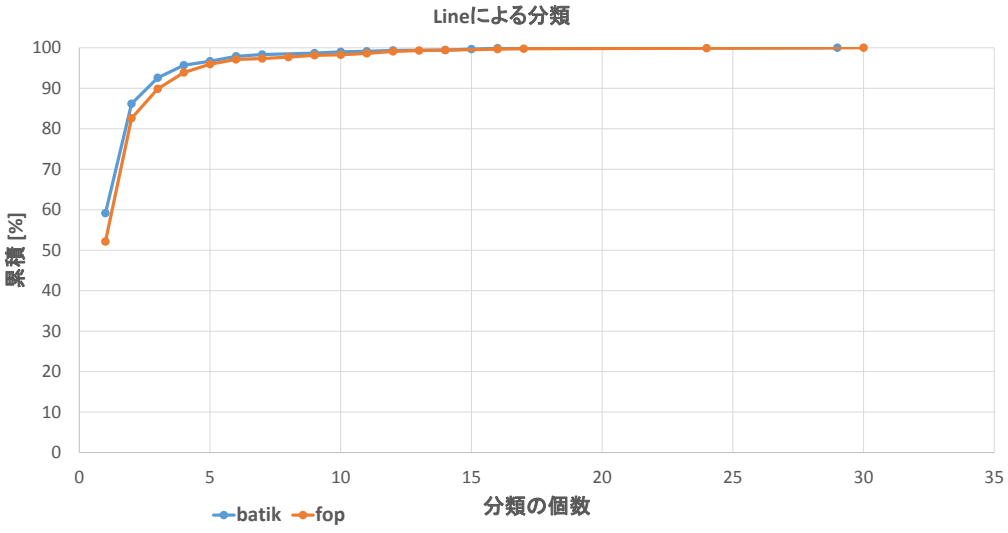

図 18: ループもしくは分岐があり，呼び出し回数 2 回以上のメソッドに対する Line による分類

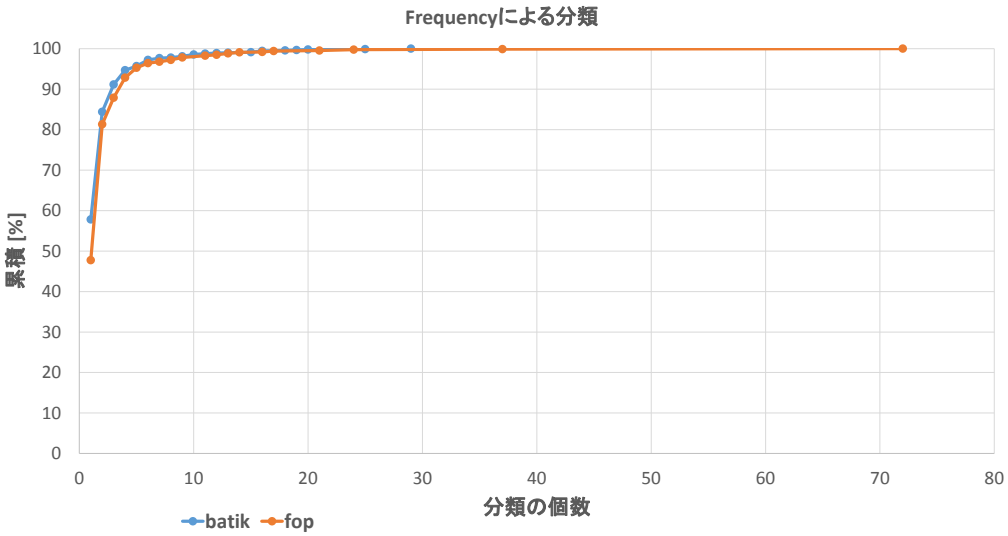

図 19: ループもしくは分岐があり，呼び出し回数 2 回以上のメソッドに対する Frequency による分類

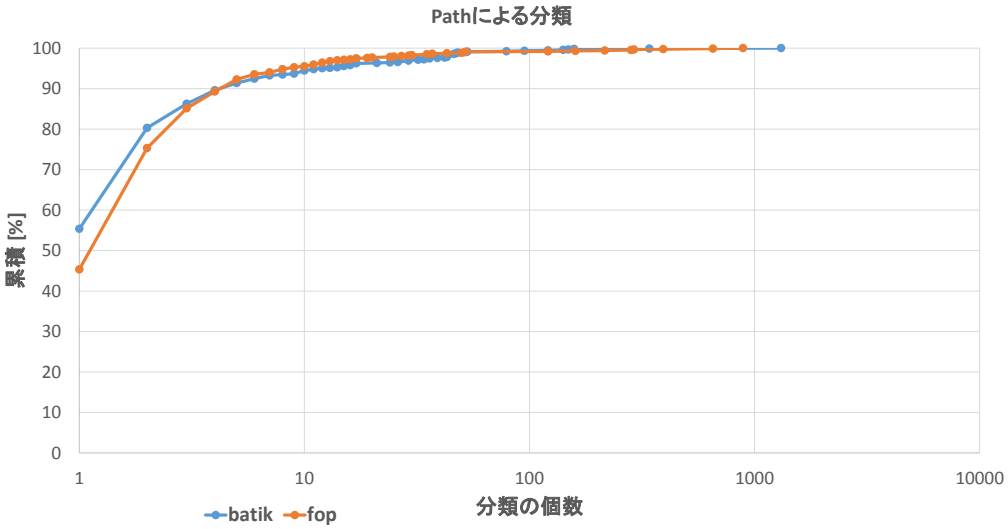

図 20: ループもしくは分岐があり，呼び出し回数 2 回以上のメソッドに対する Path による分類

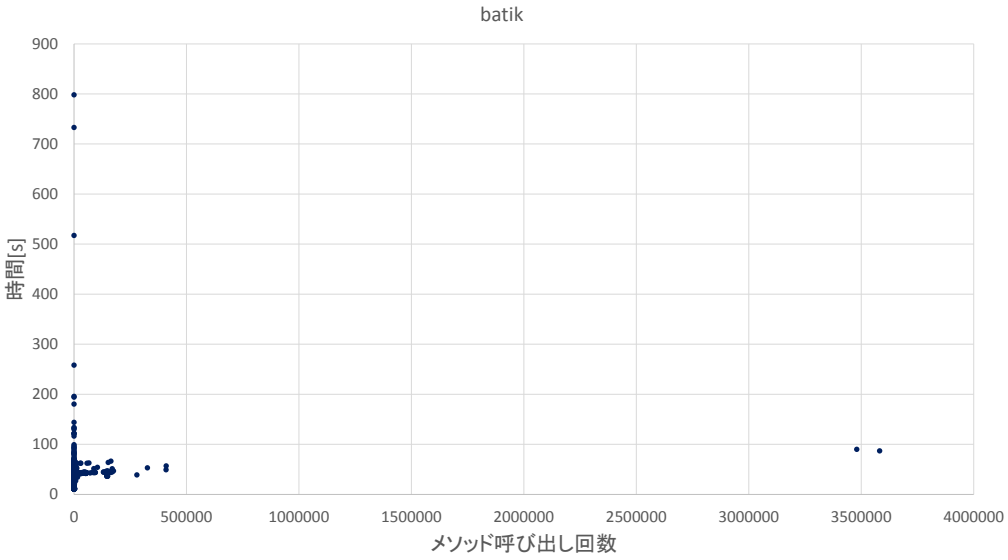

図 21: batik 内のメソッドの再現時間

5.2 実行の再現時間

本実験には，表 5 に示す環境で batik,fop 内のメソッドの再現にかかる時間を計測した．対象となるメソッド数は分類と同様，batik では 3131，fop では 3611 である．

図 21,22 が batik と fop それぞれについて，各メソッドを再現するのにかかった時間をプロットした散布図である．図では横軸にメソッドの呼び出し回数，縦軸に時間を表示している．一部のメソッドでは再現に 10 分以上かかっているが，99%のメソッドについて 100 秒以内に実行の再現が行えることから，実用に耐えうると考えられる．実行に 100 秒以上かかるメソッドは batik では 15 個，fop では 6 個であり，メソッドに対応する部分の実行ログが長いメソッドに再現時間が長い傾向が見られた．実行ログが長いメソッドでは再現に時間がかかる原因として，ログの読み出しにかかる時間が増えることが考えられる．

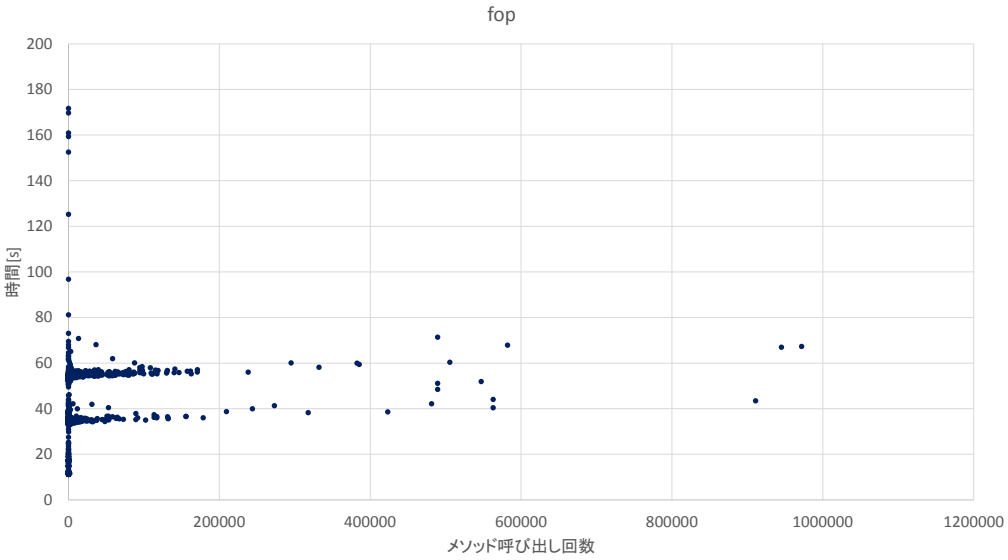

図 22: fop 内のメソッドの再現時間

6　関連研究

実行時情報に基づいた，デバッグ手法がいくつか提案されている．本節ではそれらについて述べる．

Statistical Debugging[8] は, プログラム中の欠陥の箇所を自動的に特定する Fault Localization と呼ばれる技術の 1 つである．プログラムに対して複数のテストを実行し，成功する実行と失敗する実行の実行時情報の差分から，欠陥として疑わしいプログラム文を特定することができる．しかし，この技術を用いて得られるのは欠陥がある可能性の高いプログラム文の集合であるので，開発者は疑わしいプログラム文の候補が真に欠陥かどうかを調査する必要がある．

Object-Centric Debugging[9] は，オブジェクトに着目したデバッグ手法である．開発者がしたい質問にはオブジェクトに関するものが多々あり，静的なブレークポイントでは，あるオブジェクトがいつアクセスされるかということを知りたいとき，そのオブジェクトがアクセスされ得るすべての場所にブレークポイントを設定する必要がある．この手法では，オブジェクトに対しブレークポイントを設定することで，ソースコードに多量のブレークポイントを設定することなく，オブジェクトがいつどこでアクセスされるのかが判断できる．

Relative Debugging[10] は,2 つのプログラムに同じ入力を与えて実行を比べることで欠陥を特定する技術である．開発者が 2 つのプログラム間で状態が等しくなるべき地点を指定すると，状態が異なっていた場合にそのことが通知される．この技術はソフトウェア開発などで，正しく動作していた旧バージョンのプログラムに改変を加えたところ，新バージョンでは正しく動作しなくなった場合などに，旧バージョンと新バージョンの実行を比べることで，効率的なデバッグが可能となる．

7　まとめと今後の課題

本研究では，複数回到達しているメソッドにおける実行時情報の閲覧には，繰り返し操作・閲覧が必要であるという問題に対して，実行履歴の一覧を提示する手法を提案した．そして，一覧の提示では実行履歴が膨大な量となることが予測されるため，実行経路による分類を行うことを提案し，これらの機能を持つツールを実装した．

ツールの適用可能性についての実験を行い，90%のメソッドに対し Path による分類で 5 つ以下の分類数となることがわかり，1 つの画面内に表示できることを確認した．

今後の課題としては，ツールの機能強化が挙げられる．本研究で作成したツールは 1 つのメソッドしか対象していないため，複数のメソッドについて実行履歴を閲覧したい場合にはツールを複数起動しなければならず，複数起動したツール間ではメソッド呼び出しに関する時間的な対応関係がとれていないので，その点を改良する必要があると考えられる．

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鹿島悠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] Girish Parikh. Making the immortal language work. *International Computer Programs Business Software Review* , Vol.7, No.2, 1987.

[2] R. Fjeldstad and W. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings of GUIDE 48*, 1979.

[3] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. IEEE Transactions on Software Engineering, Vol.34, No.4, pp.434-451, 2008.

[4] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software?. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE 2012), pp.255-265, 2012.

[5] GDB. http://www.sourceware.org/gdb/.

[6] Bil Lewis. Debugging Backwards in Time. In *Proceedings of the 5th International Workshop* on Automated and Algorithmic Debugging (AADUBUG 2003), pp.225-235, 2003.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovi, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pp.169–190. 2006.

[8] Alice X. Zheng , Ben Liblit , Michael I. Jordan , and Alex Aiken. Statistical Debugging of Sampled Programs. In *Proceedings of the 17th Annual Conference on Neural Information Processing Systems* (NIPS 2003).

[9] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-Centric Debugging. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE 2012), pp.485-495, 2012.

[10] 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.