REMViewer: 複数回実行されたJavaメソッドの実行経路可視化ツール

松村 俊徳　石尾 隆　鹿島 悠　井上 克郎

プログラムの実行時情報はプログラム理解に有用であることが知られており，Omniscient Debugging などのツールを使うことで開発者は実行の任意の時点の状態を調査することができる．これらのツールはある 1 つの時点の状態を分析するには有効だが，複数の実行経路を持つようなメソッドの動作を理解するには不十分である．そこで本論文では，REMViewer (Repeatedly-Executed-Method Viewer) という Java メソッドの複数の実行経路を可視化するツールを提案する．このツールはプログラムの実行を記録しておき，メソッドの 1 回ごとの実行経路を別々のビューとして表示するので，開発者は興味のある実行パスを選択し，それらの実行パスにおける局所変数の状態を比較するといった分析が可能となる．

The state of a program at runtime is useful information for developers to understand a program. Omniscient Debugging and logging-based tools enable developers to investigate the state of a program at an arbitrary point of time in an execution. While these tools are effective to analyze the state at a single point of time, they might be insufficient to understand the generic behavior of a method which includes various control-flow paths. In this paper, we propose REMViewer (Repeatedly-Executed-Method Viewer), or a tool that visualizes multiple execution paths of a Java method. The tool shows each execution path in a separated view so that developers can firstly select actual execution paths of interest and then compare the state of local variables in the paths.

1 はじめに

プログラム理解において，開発者は対象プログラムのソースコードを読むだけでなく，デバッガを用いてプログラムの実行時の状態を分析している [12] [14]. GDB や Eclipse JDT などの広く使われている対話的デバッガでは，開発者はソースコード内の興味のある部分にブレークポイントを前もって設定し，プログラムを実行する必要がある [11]. 一般的なデバッグ作業であれば，開発者はソースコードの構造を十分に知っていることが多く，このことは問題にならない．一方，他人が記述したプログラムの挙動をこれから分析しようとしている開発者にとっては，プログラムの構造や振舞いを把握する前にデバッガを的確に用いることは難しい．

プログラムの実行時の状態を分析する手法には様々なものがあるが，事前のソースコードの調査を必要としない技術としては Omniscient Debugging をはじめとする “logging-based” 技術 [6] [7] [8] が挙げられる．たとえば Omniscient Debugging [8] はプログラム実行中のすべての命令の実行順序と変数の状態を記録しておき，プログラムの実行の任意の時点の状態を計算機上で再現，分析する技術である．このような技術は実行時情報を記録するのでプログラムの実行速度は低下するが，ソフトウェア保守においてプログラムの動作を理解する上で有望なアプローチであるとされている．

プログラムの実行時の状態を再現する手法は既に実現されているが，その再現結果をプログラム理解に活用する方法については，十分な研究はなされていない．デバッグ作業では，バグによって生じた障害

REMViewer: A Visualization Tool for Multiple Execution Paths of Java Method

Toshinori Matsumura, Takashi Ishio, Yu Kashima, Katsuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

の原因を究明することが目的となっているため，プログラムの実行のある一時点（バグによって発生した障害を観測した時点）を調査の起点として，そこに至った実行経路を分析することに主眼が置かれている [7]．しかし，メソッドの一般的な動作を理解するには，特定の時点の調査だけでは不十分な可能性がある．たとえば，メソッドの内部に複数の制御パスが存在する場合，開発者は個々の制御パスについて順番に分析を行わなければならない [13]．

本研究では，Java メソッドの複数の実行経路を可視化するツール REMViewer (**R**epeatedly-**E**xecuted-**M**ethod **Viewer**) を提案する．REMViewer はプログラムの実行を記録し，ユーザーの指定したメソッドについて，そのメソッドの実行すべてを再現し表示する．個々の制御パスに対する調査を支援するため，REMViewer はメソッドの実行を実行経路に基づき分類し，それぞれの分類の代表を可視化し表示する．可視化された実行経路を比較することで，開発者は，興味のある実行を選び，その後局所変数の状態を比較するといったことが可能となる．通常のデバッガにおけるブレークポイントと比較すると，実行完了後の調査という方法に限定するかわり，開発者が実行経路を見てから任意の時刻を選択できる点が特徴である．

本論文の主要な貢献は以下の通りである．

- Java の任意のメソッドの実行を再現するツールを実現した．メソッドの実行の再生は，ユーザにとっての対話的なツールとしての有用性だけでなく，他の動的解析手法の研究基盤としても利用可能である．
- 実行経路の分類による可視化を実現した．2 つのプログラムの実行履歴に対して適用し，多くのメソッドに有効であることを確認した．

なお，本論文は文献 [9] に対して，実行履歴の観測と再生の方法に関する実装の詳細を追加したものとなっている．

以降，2 章ではツールの機能および実装を説明し，3 章ではツールの性能，4 章では関連研究，5 章でまとめと今後の課題について述べる．

2 REMViewer

REMViewer は Java を対象とした実行トレース (execution trace) の可視化ツールである．本ツールはあらかじめプログラムの実行を観測，記録しておき，ユーザーによって指定されたメソッドの動作を再現する．ツールは *Instrumentation*, *Replay*, *Viewer* という 3 つのコンポーネントで構成されている．*Instrumentation* コンポーネントは，ユーザが指定した対象 Java プログラムのバイトコードに実行トレースを出力する命令を埋め込む．得られたプログラムに観測したいテストケースを与えて実行すると，実行トレースがファイルとして保存される．*Replay* コンポーネントは，ユーザに指定されたメソッドの実行時の状態を再現する．*Viewer* コンポーネントは，得られた再現結果を対話的に閲覧する GUI を提供する．

2.1 実行トレースの記録

REMViewer は，実行トレースとしてメソッド間のすべての制御フローとデータフローを記録しておき，メソッド間で受け渡された引数などの情報をもとにメソッド内の命令列を再度実行するという方針で設計されている．

2.1.1 実行トレースの情報

メソッド間の制御フローを表すイベントは，メソッドの開始，メソッドの終了，メソッドの呼び出し，例外の送出と捕捉である．これに加えて，メソッドの引数と返り値の受け渡し，フィールドと配列へのアクセスに使われた実際の値を記録する．

すべてのイベントはスレッド ID と命令 ID という 2 つの属性を持ち，イベントが発生したスレッドとバイトコード命令を識別する．スレッド ID は，プログラムの実行時に，最初に出現したスレッドを 0 番として，実行トレースへの出現順に割り当てられる数値である．命令 ID は，ログ記録用命令を埋め込む段階で，観測対象となるバイトコードの各命令に対して数値を一意に割り当てたものである．各イベントは，表 1 に示すように，それぞれのイベントで実際に使用されたデータを保持する．たとえば，オブジェクトのフィールドを更新するイベントでは，フィール

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	仮引数の値
NormalExit	メソッドの正常な終了	返り値
ExceptionalExit	メソッドの例外発生による終了	例外オブジェクト ID
Call	メソッドの呼び出し	呼び出しの実引数の値
ReturnValue	メソッドの返り値の受け取り	呼び出し結果の返り値
GerField	フィールドの読み出し	読み出し対象オブジェクトの ID, 読み出した値
PutField	フィールドへの書き込み	書き込み対象オブジェクトの ID, 書き込んだ値
ObjectCreation	オブジェクトの生成	生成されたオブジェクトの ID
NewArray	配列の生成	生成された配列のオブジェクト ID, 配列の長さ
MultiNewArray	多次元配列の生成	生成された配列のオブジェクト ID, 配列の長さ
ArrayLoad	配列の値の読み出し	配列のオブジェクト ID, インデクス, 読みだした値
ArrayStore	配列の値の書き込み	配列のオブジェクト ID, インデクス, 書き込んだ値
ArrayLength	配列の長さの参照	配列のオブジェクト ID, 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	捕捉したオブジェクト ID
InstanceOf	instanceof 命令の実行	対象のオブジェクト ID, 結果の boolean 値
MonitorEnter	synchronized ブロックの実行開始	同期対象のオブジェクト ID
MonitorExit	synchronized ブロックの実行終了	同期対象のオブジェクト ID

ドに新たに代入される値と，代入の対象となったオブジェクトの ID が記録される．オブジェクト ID は，スレッド ID と同様に，プログラムの実行中に登場したオブジェクトに順番に `long` 型の数値を割り当てたものである．実装では，Java 仮想マシンのガベージコレクションに影響しないように `WeakReference` を使ったオブジェクトの管理を行い，オブジェクト参照から ID へと対応付けを行っている．

マルチスレッドプログラムの実行によって生じたイベント系列は，実行トレースを書き出す処理内の `synchronized` ブロックにより逐次化される．そのため，実行トレースを書き出す処理がスレッドスケジューリングに影響を与える可能性はあるが，結果として得られたマルチスレッドの動作順序はそのまま実行トレースとして出力される．

実行トレースのファイル形式は，1 イベントあたり 30 バイトのデータを固定長で並べ，それを gzip アルゴリズムにより圧縮したものである．データの内容は，イベントの種類（2 バイト），スレッド ID（4 バイト），命令 ID（4 バイト），イベントごとの情報（20 バイト）である．イベントごとの情報は，最もデータ量の多い配列の書き込み命令に合わせており，オブジェクト ID（8 バイト），配列の添え字の整数（4 バイト），配列の値（最大で `long` 型の 8 バイト）である．メソッドの引数など個数が可変のデータは，メソッド呼び出し等の対応するイベントの直後に，固定長のデータ形式に合わせて格納する．文字列オブジェクトの内容，例外オブジェクトのスタックトレース情報などの可変長データは，それぞれ別ファイルとして出力する．固定長を採用した理由は，圧縮率の向上と，展開後のデータ系列へのランダムアクセスの実現である．圧縮後のデータ量では，1 イベントあたり最大で 6 バイト程度となっている．

イベント数は膨大となる可能性があるので，実行トレースは 1000 万イベントごとに 1 ファイルの単位で分割されて出力する．プログラムの実行時のオーバーヘッドを下げるため，対象プログラムのスレッドに加えて実行トレースを書き出すためのスレッドを追加で起動することを可能としている．バッファを複数用意し，1 つのバッファにイベント系列が蓄積されている間に，満杯になったバッファを圧縮，出力する処理を実行する実装となっている．

2.1.2　実行トレース観測命令の埋め込み

REMViewer は，ASM [†1] を使用して対象プログラムに観測用命令を埋め込む．たとえばメソッドのすべての実行開始と終了を記録するために，以下のような

†1　ASM. `http://asm.ow2.org/`

ソースコードに対応する命令を作成する．

```
try {
    Logging.recordMethodEntry(0);
    // 元々のメソッド本体の内容
    Logging.recordMethodExit(1);
} catch (Throwable t) {
    Logging.recordExceptionalExit(2, t);
    throw t;
}
```

ここで，Logging というクラスが実行トレースの観測用クラス，各メソッドの引数に登場する定数が命令 ID である．Replay コンポーネント等は命令 ID から対象のバイトコード位置を特定する．

バイトコードの中に命令を埋め込む都合上，記録にはいくつかの制限がある．

- メソッドに含まれる命令数が非常に多い場合，追加の命令を埋め込んだ結果，メソッドの大きさが Java 仮想マシンの制限（64KB）を超えてしまうことがある．たとえば配列を定数の列で初期化しているメソッドは，配列の各要素への代入の並びとしてコンパイルされているため，各代入に対応した観測命令を追加することになり，このサイズ制限に抵触することがある．この制限に抵触した場合，REMViewer はまず配列への代入命令のみを観測対象から除外し，それでも制限に抵触する場合はメソッドの開始と終了のみを観測する最小限の命令を埋め込む．
- Java のクラスは初めて使用される際に ClassNotFoundException が発生し，クラスの読み込みと初期化が実行されるが，実行トレースの記録を行うために使われる java.io パッケージのクラスについては，この初期化の発生を記録することができない．
- バイトコードの改変を禁止する署名付きプログラムに対しては観測を行うことができない．

なお，観測範囲は，バイトコード変換をかける対象のファイルを選定することで指定可能である．各メソッドについて，メソッドの外部から受け取る引数やフィールド，例外等の情報はすべて記録されるため，観測範囲内のメソッドについて自由にその状態を再現することが可能である．

2.2 再現

Replay コンポーネントは，プログラムの実行が終わった後，前準備として一度実行トレースを探索し，すべてのメソッド開始イベントの位置を特定しておく．そして，ユーザから指定されたメソッドの開始位置から，実行トレースに記録された値を利用しながらバイトコード命令を解釈し，指定されたメソッドの実行を再現する．

メソッドの実行の再現方法を詳しく説明すると，まず実行トレースのメソッド開始イベントの位置から引数の情報を読み出し，ローカル変数の初期状態を構築する．そして，次のイベントを 1 つ読み出し，そのイベントに紐付けられた命令 ID までのバイトコード命令を順に解釈し，スタックやローカル変数の状態を更新する．配列やフィールドへのアクセスの結果など，メソッド外部の情報については読み出したイベントに保持されている値を利用する．命令 ID までのバイトコード命令の実行が終わると，実行トレース中の次のイベントを読み出し，同様にバイトコード命令を解釈していく．この操作をメソッドの終了イベントを読み出すまで繰り返すことで，メソッドの実行を再現する．

Java ではローカル変数を未初期化の状態で使うことはできないため，すべての変数はイベント情報に格納された引数の値か，バイトコード命令が参照する定数値から値を決定することができる．また，メソッドの戻り値などの情報はイベントとして記録されているため，メソッドの呼び出し先など，再生対象ではないメソッドの中での動作の再生は，すべて省略することができる．マルチスレッドで並行して何らかの処理が実行されていたとしても，それによって更新される可能性がある値は配列，フィールドに限られるため，実行トレースで実際に参照した値を使用すれば，スレッド間の影響があっても実行を正しく再現することができる．

ユーザーが指定したメソッドが N 回実行されていた場合には，N 回の実行すべてを再現する．再現されたメソッドの実行を $E = \{e_1, \cdots, e_N\}$ とおく．1

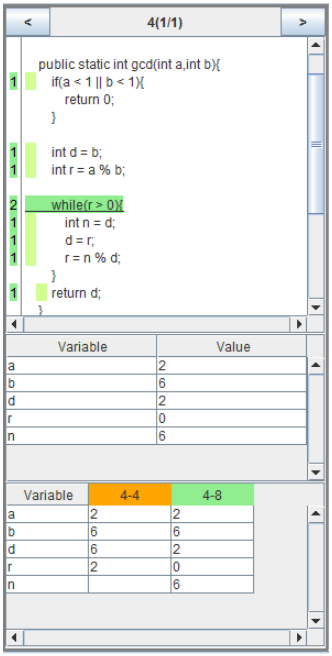

図 1　対象メソッドの 1 回の実行を表すビュー

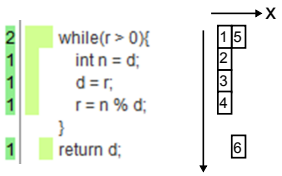

図 2　矩形により示される実行経路

回のメソッドの実行 e_i は状態 $\langle b_k, l_k, v_k \rangle$ の列であり，b_k は k 番目のバイトコード命令，l_k はバイトコード命令に対応するソースコード中の行番号，v_k は b_k を実行した後の局所変数の状態をそれぞれ表している．

2.3　可視化

REMViewer は 1 回のメソッド実行に対し，図 1 のようなビューを表示する．なお，図 1 に表示されている gcd という名前のメソッドは，引数の 2 つの正整数の最大公約数を求めるメソッドであり，ユークリッドの互除法のアルゴリズムに従って実装されている．

ビューは，上から順にソースコード，変数の状態，行における変数の状態という 3 つの領域から構成されている．ソースコードの表示領域は，デバッガと同様，実行の位置を指すカーソルを持っている．ユーザーはこのカーソルを実行経路に沿って前後に操作することができ，現在の実行位置における変数の状態がビュー中央の領域に表示される．これに加えて，ユーザーはソースコード中の任意の行をクリックし選択することで，その行における変数の状態を確認することが可能である．選択した行における変数の状態はビューの下部に表示される．このとき，たとえばループ内の行を選択した場合，複数回その行が実行される可能性があるため，選択した行における変数の状態は表形式での表示となっている．図 1 の例では，2 つの列がそれぞれ while 文の 1 度目の実行と 2 度目の実行を示している．列名の 4-4, 4-8 という表示は，対象メソッドの 4 回目の実行における 4 ステップ目，8 ステップ目の変数の状態であることを示している．

実行経路を可視化するため，ソースコードの背景には緑色の矩形を表示している．図 2 は実行経路の例を示しており，実行 e における各状態 $\langle b_k, l_k, v_k \rangle$ が座標 (x_k, l_k) の矩形として描かれている．水平方向の値はプログラムの実行がソースコードの上方向に向かうときにインクリメントされる．すなわち，$l_{k-1} > l_k$ であれば $x_k = x_{k-1} + 1$，そうでなければ $x_k = x_{k-1}$ となる．

メソッド実行の比較を支援するため REMViewer では複数のビューを同時に表示する．ここで，すべてのメソッド実行を単純に並べて表示すると表示領域が不足するため，異なる実行経路をたどる実行を優先的に表示する．そのために，REMViewer は実行 E をグループに分類し，グループから 1 つずつメソッド実行を表示する．REMViewer は path-based と line-based という 2 つの分類基準を採用しており，それぞれ，実行の集合 E を同値類に分類する．path-based の分類では，集合 E を実行したバイトコード命令番号の列が等しい同値類 $P(E) = \{P_1, \cdots, P_{N_p}\}$ に分類する (N_p は同値類の個数)．$Path(e)$ を実行 e 中のバイトコード命令番号の列 $(b_1, b_2, \cdots)$ とすると，分類は以下のような条件で表現できる．

$\forall e_1 \in P_i, e_2 \in P_j.\ \ i = j \Leftrightarrow Path(e_1) = Path(e_2)$

同様に，line-based の分類では，実行 E をバイト

コード命令番号の集合が等しい同値類 $L(E) = \{L_1, \cdots, L_{N_l}\}$ に分類する (N_l は同値類の個数).
$\forall e_1 \in L_i, e_2 \in L_j.\ \ i = j \Leftrightarrow Lines(e_1) = Lines(e_2)$
ここで $Lines(e)$ は実行 e 中で実行されたバイトコード命令番号の集合である.

REMViewer は分類された各グループから代表を 1 つずつ選びだし表示を行う. 図 3 が Viewer のスクリーンショットである. 図 1 のメソッドを対象に 3 つの実行経路を含むビューを表示している. 左のビューは if 文と while 文が共に false となる実行経路, 中央のビューは if 文内の return 文を通る実行経路, 右のビューは while 文内のブロックを 1 度だけ実行する実行経路をそれぞれ示している. これら 3 つのビューでメソッドの基本的な動作を特定し, ユーザーは実行経路や変数の状態を 1 つのウィンドウで比較することができる. 各ビューの上部にはそれぞれのグループに含まれる実行の個数が表示されるので, 珍しい実行とそうでない実行を区別することも可能である. 同一グループに分類された実行経路であっても, たとえば while 文内のブロックが実行された回数などは異なる可能性があるので, ユーザは各ビューに表示する実行パスを切り替えたり, グループ化を解除することで, 詳細な動作の比較を行うことも可能としている.

3 ツールの性能

実行時間と可視化の適応可能性という 2 つの観点でツールの評価を行った. 実験対象としては, Java アプリケーションのベンチマーク集合である DaCapo benchmark [4] に収録された 2 つのアプリケーション, batik と fop を使用した. DaCapo benchmark のアプリケーションは実行のサイズを指定できるが, ここでは, default を指定しアプリケーションの実行を記録した. 実行環境としては, CPU が Intel Xeon 2.90Hz, メモリが 256GB であるマシン上で行っている.

3.1 実行時間

表 2 に実行トレースに関する情報を示す. 最初の 2 行は, 実行トレースを記録しない通常の各アプリケーションの実行時間と, 実行トレースの記録を行った場合の実行時間を示している.「履歴サイズ」は実行トレースとして保存されたファイルの合計サイズであり,「メソッド数」はアプリケーション内部で少なくとも 1 回は実行されたメソッドの数,「メソッド実行数」はメソッドの総実行回数をそれぞれ示している.

表 2　実行トレース

Benchmark	batik	fop
通常の実行時間	4.44 sec	2.80 sec
トレース記録の実行時間	33.67 sec	36.91 sec
履歴サイズ	717MB	860MB
メソッド数	3,131	3,611
メソッド実行数	17,319,055	26,551,919

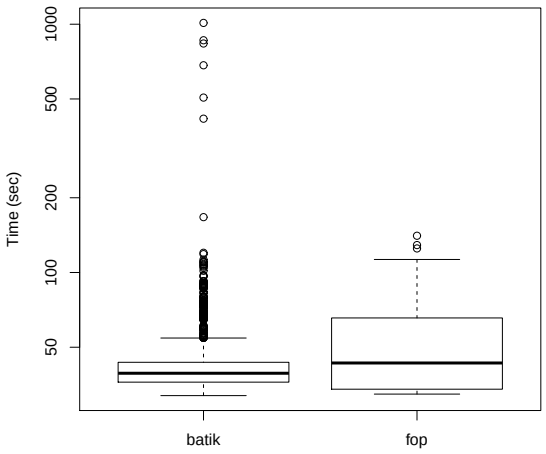

図 4　1 メソッドの再現に要する時間の分布

対象となるプログラムの実行に関して詳細な実行トレースを記録しているため, プログラムの実行時間は大きく増加しているが, 開発者がいくつかの機能を試験的に実行する程度であれば, 開発者の作業を大幅に中断させるほどのコストではない. また, 一度トレースを記録すれば, その実行に関してはすべてのメソッドを調査することができるため, あらかじめテスト自動化のようなツールを活用して事前にトレースを収集しておくことも可能である.

図 4 はメソッドの再現にかかる時間の分布を箱ひげ図で示したものである. batik では平均で 44.7 秒,

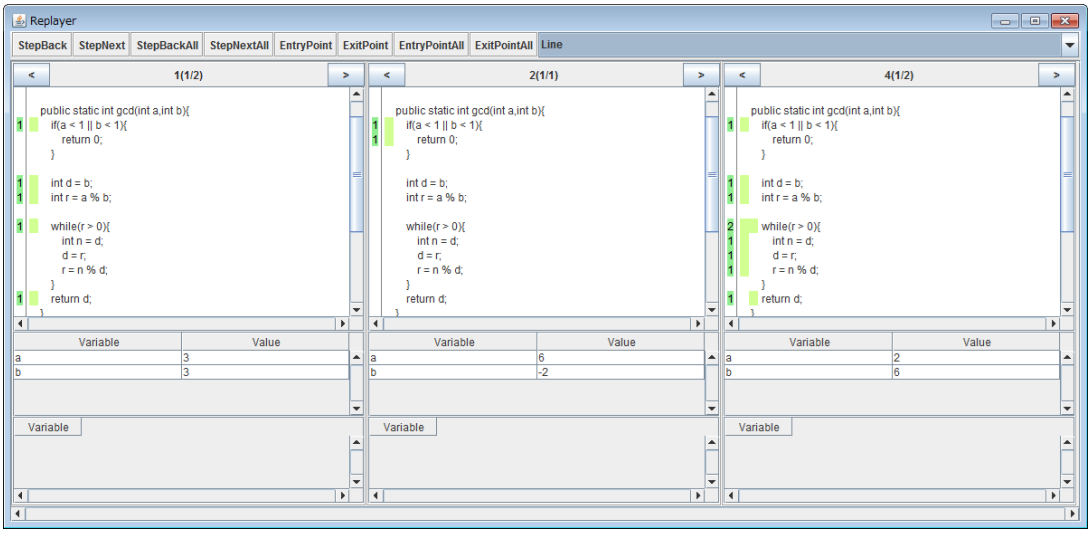

図 3　対象メソッドの 3 つの実行経路を表示している REMViewer のスクリーンショット

表 3　分類対象としたメソッドの情報

Benchmark	batik	fop
メソッド数	1029	1160
メソッド内の分岐数の平均	5.05	4.11
メソッドの実行回数の平均	8512	8939

fop では平均で 50.6 秒の時間がかかっている．開発者は一般に複数のメソッドを調査することが想定されるが，1 つのメソッドについて調査している間に他のメソッドの再生の計算を行うといった工夫により，利用者の待ち時間の合計は削減することが可能であると考えている．

3.2　分類

頻繁に実行されるメソッドであっても可視化が適用可能であるかを調査するため，各メソッドの実行経路の分類によって得られるグループ数を調査した．条件分岐が含まれないメソッドは path-based, line-based のいずれであっても常に 1 つのグループに分類されるので，条件分岐を少なくとも 1 つ含むメソッドを計測対象とする．batik では 1029，fop では 1160 個のメソッドに対して分類を行った．対象としたメソッドに関する情報を表 3 に示す．

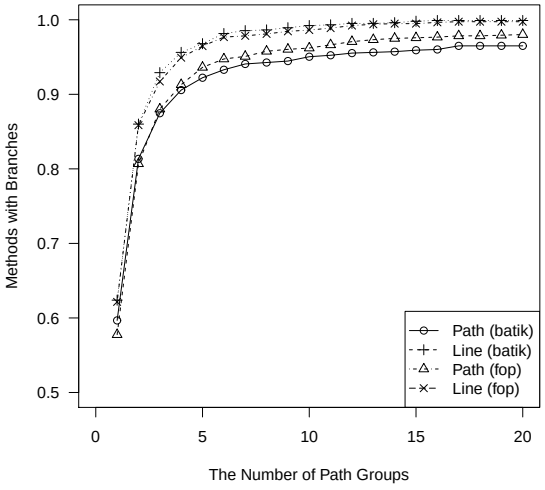

図 5　分類数の累積相対度数グラフ

図 5 はそれぞれの分類基準における分類数に関する累積度数グラフである．グラフから path-based の分類では 60%のメソッドが 1 つのグループに分類されることが分かる．言い換えれば，これら 60%のメソッドは 1 つの実行経路しか実行されていない．90%のメソッドが line-based では高々3 個，path-based では高々5 個のグループに分類されている．この結果から，多くのメソッド実行は少数のグループに分類され

ることが分かる．

一方で，分類数が多くなるメソッドも存在した．`org.apache.fop.layoutmgr.BreakingAlgorithm` クラスの `considerLegalBreak` メソッドは，全メソッド中 line-based による分類数が最大の 30 であった．このメソッドには 28 個の分岐が含まれており，このような複雑なメソッドに対しても分析を可能にするためには分類方法を改善する必要がある．たとえば，実行を通常の実行と例外的な実行に分類することで，開発者はまず通常の実行におけるメソッドの振る舞いを調査するといったことが可能になると考えられる．

4 関連研究

プログラムの実行を過去にさかのぼって調査することはデバッグに非常に有用である．Ko らは制御およびデータの依存関係をさかのぼる手法 [7] を，Mirghasemi らは変数が代入された位置にさかのぼる手法 [10] をそれぞれ提案している．これらの手法はデバッグを主眼としており，開発者が調査したい特定の時刻を選択できることを前提としている．我々のツールは，開発者がソースコードを読解する際，まず対象のソースコードを選定してから興味のある実行経路，そして時刻の順で選択できるようにしている．

Joshi らは指定されたメソッドの実行の再現を行う Selective Capture and Replay 技術を提案し，開発者の作業における有効性を確認している [6]．Omniscient Debugging [8] のような完全なプログラムの状態の再現は，対象プログラムのメモリ状態を再現するために必要なメモリ使用量と実行トレースの解析時間の両面で現実的とは言い難いが，Selective Capture and Replay は再現対象のメソッドを限定することで実用性を達成している．我々のツールもまたメソッド単位の実行を再現することで，実用的なサイズのプログラムであっても状態の分析を可能とし，さらに複数回の実行から特定の振る舞いを選択して分析するためのユーザーインターフェースを提供した．

Jones らは複数のテストケースの実行を比較し，成功した実行と失敗した実行とを比較するツール Tarantula を提案した [5]．我々のツールはデバッグではなく，プログラム理解を目的としているため，成功と失敗の区別は行わず，実行経路による分類のみを行っている．

Abramson らはバージョンの異なる 2 つのプログラムの実行を比較する Relative Debugging を提案した [1]．我々のツールは単一バージョンのプログラムでの複数回の実行を 1 つのウィンドウで比較するものである．

Bell らは実行トレースを記録する際のオーバーヘッドを少なくする Chronicler という技術を提案した [3]．我々のツールにもこの技術を応用することで性能を向上できる可能性がある．

Alsallakh らは，プログラムのある 1 つの変数に着目し，その値の変化を可視化する手法を提案している [2]．我々のツールは選択された行における変数の値の系列を抽出できるので，この手法のような可視化を適用すると，より効果的な分析ができるようになる可能性がある．また，Sagdeo らはメソッドの特定の実行経路に対して不変条件を抽出する手法を提案しており [13]，各実行において取りうる変数の状態の要約を作る，異常値を発見するといったツールの拡張に応用できると考えられる．

5 まとめと今後の課題

我々はメソッドの実行経路および変数の状態を可視化するツール REMViewer を提案した．REMViewer を使うことで，開発者は実行経路を比較しながらメソッドの振る舞いを調査することが可能となる．本ツールを使うことで，メソッドのテストケースの効率的な理解や，デバッグにおいて異常な実行経路を発見するのに利用できると我々は考えている．また，実行の記録，再現がそのものが動的スライスの計算等の，他の動的解析技術の基盤としても利用可能である．

今後の課題としては，開発者の実際の活動における有用性の評価が挙げられる．本ツールの現在の実装では，メソッド内部で参照されたデータのみを用いて状態の再現を行うため，再生したメソッドの中で一度も参照されなかったオブジェクトのフィールドについては，再現の範囲に含まれない．開発者が作業中にどのような情報を必要とするのか，ツールの操作履歴を記録するなどして，ツールの機能を改善していく必要が

ある．

実行経路の分類の改善も今後の課題として挙げられる．実行が複雑なメソッドほど，実際の動作を分析できることに価値があると考えられるが，そのようなメソッドほど通過した命令行などによる分類では結果が細分化されやすい．メソッドの通常の振る舞いと例外的な振る舞いをより簡単に区別できるようにする必要がある．

謝辞 本研究は科研費 No.26280021 の助成を得た．

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] Bilal Alsallakh, Peter Bodesinsky, Alexander Gruber, and Silvia Miksch. Visual tracing for the eclipse java debugger. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pp. 545–548, 2012.

[3] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of International Conference on Software Engineering*, pp. 362–371, 2013.

[4] 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 International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 169–190, 2006.

[5] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of International Conference on Software Engineering*, pp. 467–477, 2002.

[6] Shrinivas Joshi and Alessandro Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *Proceedings of International Conference on Software Maintenance*, pp. 234–243, 2007.

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

[8] Bil Lewis. Debugging backwards in time. In *Proceedigns of International Workshop on Automated Debugging*, 2003.

[9] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proceedings of International Conference on Program Comprehension*, pp. 253–257, 2014.

[10] Salman Mirghasemi, John J. Barton, and Claude Petitpierre. Querypoint: Moving backwards on wrong values in the buggy execution. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pp. 436–439, 2011.

[11] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceedings of International Conference on Software Engineering*, pp. 485–495, 2012.

[12] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of International Conference on Software Engineering*, pp. 255–265, 2012.

[13] Parth Sagdeo, Viraj Athavale, Sumant Kowshik, and Shobha Vasudevan. Precis: Inferring invariants using program path guided clustering. In *Proceedings of International Conference on Automated Software Engineering*, pp. 532–535, 2011.

[14] 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.