修士学位論文

題目

蓄積されたオブジェクトの動作履歴を用いた実行履歴削減手法の提案

指導教員
井上 克郎 教授

報告者
脇阪 大輝

平成 26 年 2 月 5 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 25 年度 修士学位論文

蓄積されたオブジェクトの動作履歴を用いた実行履歴削減手法の提案

脇阪 大輝

内容梗概

ソフトウェアは，リリース前にテストが実施され，そのときに含まれる欠陥は発見され修正される．しかし，すべての欠陥が発見されるとは限らず，欠陥が残存したまま，リリースされることもある．このような欠陥は，ソフトウェア利用者のもとで障害を引き起こし，初めて顕在化する．この欠陥を修正するために，開発者はまず起こった障害を開発環境で再現することを試みる．その再現のためには，プログラムの動作を詳細に記録した実行履歴が有用である．しかし，プログラムの動作情報を詳細に含んだ実行履歴は，そのデータ量が膨大となり，ソフトウェア利用者の環境のもとでそれを保存することは現実には好ましくない．

そこで本研究では，プログラムが開発期間中には観測されていない動作をしそうなときにのみ，実行履歴を記録することによって，実行履歴の量を削減する手法を提案する．提案手法では，プログラムを事前に何度か実行しておき，そのときのオブジェクトの動作を Dynamic Object Process Graph（DOPG）として抽出する．利用者の環境での実行では，DOPG をオートマトンとみなして操作することにより，出現したオブジェクトの動作が事前の実行でのオブジェクトの動作と一致しているかを判定し，不一致であるとみなされたオブジェクトが入力となっているメソッドの実行のみに対して，実行履歴の記録を行う．また，評価実験では，DaCapo ベンチマークに含まれる 4 つのアプリケーションについて，小規模の実行を事前の実行として，大規模な実行を利用者の環境での実行として用いることで，実行履歴の記録量を減らすことができるかどうかの調査を行った．その結果から，実行履歴を 22.8%から 67.23%の量に減らすことができることを確認した．

主な用語

デバッグ
動的解析
実行履歴

目 次

1　はじめに

ソフトウェアはリリースされる前の開発期間中に，静的解析や動的解析技術を利用したテストを実施される．そのときに，ソフトウェアに含まれる欠陥は発見され，修正される．しかし，すべての欠陥が，リリース前のテストによって発見されるとは限らない．たとえば，プログラムのテストが行われた度合はカバレッジによって計算されるが，複合条件網羅（multiple condition coverage）や経路組合せ網羅（path coverage）などの基準を用いたカバレッジが 100%になるまでテストが必ずしも行われるわけではない．また，命令網羅（statement coverage）で 100%のカバレッジ率に到達するまでテストを行うことは可能であるが，ソフトウェアに欠陥がないことを強く保証できるほど欠陥検出力があるわけではない．このような完全でないテストによって，欠陥が残存したままのソフトウェアがリリースされることが多々ある [3, 9]．欠陥を含んだソフトウェアは，ときに本番環境での稼働時に，その欠陥によって障害を引き起こす．

そのような障害を引き起こした欠陥を修正するために，開発者は，まず生じた障害を開発環境で再現することを試みる．障害を再現するための手掛かりとして，ソフトウェア利用者の証言や，障害発生時に出力されるエラーログなどが用いられる [17]．そのような情報だけでは欠陥の特定が困難である場合には，プログラムの動作についてのより詳細な情報を含む実行履歴が必要とされる．実行履歴を用いて，障害発生時のプログラムの内部状態や実行経路を再現することで，障害の原因となる欠陥箇所の特定を容易にする．

このようなリリース後に発生する障害を開発環境で再現できるようにするために，プログラムの実行履歴を記録しながら，本番環境でプログラムを実行する [10]．しかし，プログラムの動作を詳細に再現することが可能な実行履歴は，そのデータ量が非常に大きくなる．したがって，利用者の環境で動作するプログラムから，詳細な動作の情報を含む実行履歴を記録することは現実的には好ましくない．すべてのプログラムの実行履歴を記録するのではなく，必要のない記録をせずに実行履歴の量を減らすことが求められる．

本研究では，メソッドの入力となるオブジェクトの動作が同一であれば，そのメソッドの動作も同一であると仮定し，入力となるオブジェクトが既知の動作をしているメソッドの実行の実行履歴を記録しないことで，実行履歴の量を削減する手法を提案する．提案手法では，対象のプログラムをあらかじめ実行し，その実行において出現したオブジェクトの動作と，実行されたメソッドとその入力となったオブジェクトの組を蓄積する．その後，本番環境でプログラムを実行するとき，その実行で出現したオブジェクトの動作と，あらかじめ蓄えておいたオブジェクトの動作を比較する．比較した結果，オブジェクトの動作が，蓄えたオブジェクトの動作と異なっているとき，そのオブジェクトは未知の動作をしているとみなす．提案手法では，この未知の動作をしているオブジェクトを入力に含むメソッドを実行履歴の

記録対象とする．

提案手法を評価するために行った評価実験では，実行履歴中に含まれるイベントのうち，提案手法によってどの程度の数を記録するだけで済むかを計測した．また，プログラムの事前の実行では観察されなかった動作をするメソッドの実行を，提案手法によって実行履歴の記録対象にできているかを調査した．実験の結果から，4 つの Java アプリケーションに対して，実行履歴を約 22.8%から 67.23%の量に減らすことができた．また，2 つのアプリケーションでは，事前の実行には観察されなかった動作をするメソッドの実行の約 9 割を実行履歴記録対象とすることができていた．

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．そして，3 章で提案手法である実行履歴削減手法の説明をし，4 章では実施した評価実験について説明する．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

プログラムの欠陥は，開発期間中に，静的解析や動的解析を用いたテスト [6, 11] によって，ある程度の数が発見され，修正される．しかし，すべての欠陥を修正することができるとは限らず，欠陥を含んだままのプログラムがリリースされることは多々ある．本章では，研究の背景として，実行履歴を用いたプログラムのデバッグ手法が有効であることを紹介し，リリース後のプログラムに生じる障害を，開発環境で再現するためには，そのようなデバッグ手法が使えなくなる問題について説明する．また，リリース後の障害を再現する実行履歴を減らすための既存の研究と，本研究のアプローチのもととなっている，実行履歴の比較を行う研究を紹介する．

2.1　実行履歴を用いたデバッグ

プログラムの実行履歴は，プログラムがどのような動作をしたかを記録したもので，プログラムの動作を表すイベントが時系列順に並んでいるものである．実行履歴の形式は，プログラムの欠陥修正に用いる方法に対して，適当なものが選ばれる．たとえば，単純にプログラムの異常終了した位置をプログラムの修正に用いる場合であれば，プログラム終了時に実行中であったメソッドを示すスタックトレースを実行履歴として用いる．

Omniscient Debugging[12] は，あるプログラムの実行から取得した実行履歴を用いて，後からそのプログラムの実行の，任意の状態を再現することができる．また，任意の状態からプログラムの実行を進めたり，逆に戻すことが可能である．このようなデバッグ手法を実現するためには，プログラムの動作を詳細に記録した実行履歴が必要となる．たとえば，メソッドの呼出しやオブジェクトの生成，フィールドの読み書き等を含んだものである．このデバッグ手法を用いることによって，過去のプログラムの実行を，いつでも後から再現することが可能となり，障害の再現や，欠陥の特定と修正を行う開発者を支援することができる．並列プログラムのデバッグにおいて，Omniscient Debugging のような再現技術が，有効でないこともあると指摘されているが [13]，そうではないプログラムのデバッグにおいては，このようなデバッグ手法が有効であるとされている [14]．本研究においても，Omniscient Debugging が有効であるようなプログラムのデバッグを支援することを想定している．

リリース後のプログラムに障害が生じたとき，その原因となる欠陥を発見するためにも，実行履歴が利用される．プログラムの利用者が，プログラムを実行すると同時に，実行履歴を取得しておき，障害が生じたときにその実行履歴を開発者が確認し，欠陥の特定に役立てる．しかし，実行履歴の記録は，プログラムのパフォーマンスに悪影響を及ぼす．それはたとえば，プログラムの実行速度の著しい低下であったり，使用ディスク領域の増大である．このようなパフォーマンスへの悪影響を考慮しなければならない場合，大規模な実行履歴が

必要となる Omniscient Debugging のような手法による利用者環境でのプログラム動作の再現は利用できず，実行履歴の量を少量にとどめる取り組みが行われている．

2.2 実行履歴削減の取組み

実行履歴の量を小さくするために，プログラムへの外部入力など，実行を再現するために最低限必要な動作情報のみを，実行履歴として記録する手法がある [1, 3, 18]．このような手法は，プログラムの動作情報を実行履歴から読み込み，それを用いてプログラムを実行することで，プログラム利用者の環境での実行すべてを再現しようとしている．たとえば，プログラムへの外部入力のみを実行履歴として記録しているとすると，プログラムを実行する際に，プログラムへの外部入力を実行履歴に置き換える．これにより，プログラムの実行を再現でき，プログラムの動作すべてを詳細に記録した場合と比較して，実行履歴の量を減らすことができる．しかし，このような手法は，プログラムの内部状態を再現するために，プログラムを最初から実行する必要がある．したがって，実行時間が非常に長いようなサーバーアプリケーション等の実行を再現することには不向きである．提案手法では，メソッド実行の単位で実行履歴を記録するため，プログラム実行の部分的な再現に対応している．

また，プログラムに障害が発生してから，その原因となる欠陥を含んでいそうなプログラムの箇所を推定し，その部分だけから実行履歴を取得する手法がある [2]．プログラム全体に対して実行履歴を記録する場合に比べると，実行履歴を小さくすることができる．この手法では，初めてプログラムに障害が発生してからも，欠陥の場所を推定するために，欠陥を含むプログラムを稼働させ続ける必要がある．提案手法では，プログラムに初めて障害が生じたときには，既にそのときの実行履歴が記録できていることを目指している．

実行履歴に対してファイル圧縮技術 [7] や，実行履歴用の圧縮技術 [8] を適用することで，ファイルサイズを小さくすることも行われる．これで実行履歴のデータサイズを小さくすることが可能である．圧縮技術を用いた場合，その実行履歴を利用する際には，その実行履歴の復号化を行う必要がある．また，提案手法によって得られる実行履歴にも，ファイル圧縮技術を適用することは可能である．

2.3 実行履歴の比較の取組み

本研究のアプローチのもととなっている，実行履歴の比較を行っている事例を紹介する．

宗像らの研究 [15] では，プログラム中に出現するオブジェクトに関しての履歴をオブジェクト間で比較することによって，同じクラスに属するオブジェクトであっても，その動作は少数のグループに分類できることを明らかにした．オブジェクト指向プログラムの実行は，それに関わるオブジェクトの動作によって進行する．本研究では，オブジェクトの動作が同

じグループに分類される場合，それらのオブジェクトが関わるプログラムの実行もまた，同種の動作になると仮定した．

Dallmeier らの研究 [5] では，プログラムの passing run と failing run の実行履歴から抽出したオブジェクトの動作モデルを比較することによって，プログラムに含まれる欠陥に対する修正案を自動生成することに成功している．本研究でも，プログラムに障害が生じるときには，その実行に関わるオブジェクトが，過去には観測されていない動作をしている可能性があることに着目し，オブジェクトの動作が異なっている部分のプログラム実行を実行履歴の記録対象としている．

3　実行履歴削減手法

本研究では，Java プログラムを対象とし，その実行履歴の量を減らすために，未知の動作に着目して，実行履歴を取得する手法を提案する．提案手法では，プログラムを事前に実行しておき，そのときに出現したオブジェクトの動作を抽出する．そして後で実行履歴を取得しながらプログラムを実行するときには，そこで出現するオブジェクトの動作と，事前に抽出したオブジェクトの動作を比較する．比較により，事前の実行時と違う動作をしているオブジェクトを見つけ，そのようなオブジェクトが入力となるメソッド実行を実行履歴の取得対象とする．

提案手法は，事前にプログラムを実行するときの処理と，後で実行するときの処理で構成される．以下にその処理の概要を示す．

事前の実行 プログラムの簡易な実行

- 出現したオブジェクトの動作を抽出する
- 実行されたメソッドとその入力となったオブジェクト集合との組を記録する

後の実行 詳細な実行履歴を取得しながら行う実行

- 出現したオブジェクトの動作と，事前の実行でのオブジェクトの動作を比較する
- メソッド実行開始時に実行履歴取得対象かどうかを判定する

本章では，提案手法に用いる実行履歴と，これらの処理について説明する．

3.1　提案手法で用いる実行履歴

提案手法では，対象のプログラムを実行するときに，実行履歴を取得する必要がある．この実行履歴は，以下のイベントを少なくとも含んでいる必要がある．

Call $\langle t, thread, o, m, l\rangle$．メソッド呼出しを表す．

Entry $\langle t, thread, m\rangle$．メソッド実行の開始を表す．

Exit $\langle t, thread, m\rangle$．メソッド実行の終了を表す．

Return $\langle t, thread, m\rangle$．メソッド呼出しからの復帰を表す．

Parameter $\langle t, thread, o, P\rangle$．Entry イベントに対応し，そのメソッド実行のパラメータの値を表す．

```
1: public static void main(String[] argv){
2:    String exp = argv[0];
3:    Stack stack = new Stack();
4:    for(int i=0;i<exp.length();i++){
5:       char c = exp.charAt(i);
6:       if(isOperand(c)){
7:          stack.push(Character.digit(c, 10));
8:       }else{
9:          calc(stack, c);
10:      }
11:   }
12: }
```

```
20:private static void calc(Stack stack, char operator){
21:   switch(operator){
22:   case '+':
23:      stack.push(stack.pop() + stack.pop());
24:      break;
25:   case '*':
26:   …
27:   }
28:}
```

図 1: 例として用いるプログラム

すべてのイベントに含まれる $t, thread$ は，それぞれ，イベントが発生した順序を表すタイムスタンプと，イベントが発生したスレッドの ID である．o はメソッド呼出しやその実行におけるレシーバオブジェクトの ID である．m は呼出されたまたは実行されたメソッドを示す．l は実行されたメソッド呼出し命令のプログラム中での位置を一意に示す ID である．$P = \{p_1, \ldots, p_n\}$ はメソッド実行に与えられた引数の値の列である．ここで，図 1 のようなプログラムを例に用いる．このプログラムは，コマンドラインから与えられた逆ポーランド記法の数式を計算するものである．引数に数式 "1 2 +" を与えたとすると，被演算子である 1，2 が順にスタックへ格納され，その後，calc メソッドの中で足し算が行われ，その結果がスタックへ格納される．このときに得られる実行履歴の一部が表 1 のようになる．タイムスタンプ t の値によって，各イベントの発生順序が示される．また，例中では，static メソッドの実行におけるレシーバオブジェクト o を None と表記し，命令位置 l を図 1 の行番号を用いて表記している．

3.2 事前の実行

事前にプログラムを実行して実行履歴を取得し，オブジェクトの動作とメソッドの実行のパラメータについての情報を収集する．オブジェクトの動作の抽出では，Call イベント，Exit イベント，Return イベントを用いる．メソッド実行とその入力となったオブジェクト集合の取得には，Entry イベントと，Parameter イベントを用いる．

3.2.1 オブジェクトの動作の抽出

実行履歴を用いて，オブジェクトごとの動作を抽出する．オブジェクトの動作は，Dynamic Object Process Graph（DOPG）[16] として抽出する．DOPG は，オブジェクトに対するメソッド呼出しの順序を示す，動的に生成される有向グラフである．1 つのオブジェクトごとに 1 つの DOPG が作成される．本手法で用いる DOPG は，以下の種類の頂点をもつ．

表 1: 実行履歴の例

イベント	t	o	m	l	P
Entry	1	None	main		
Parameter	2	None			0
Call	3	2	Stack	3	
Return	4	2	Stack	3	
Call	5	1	length	4	
Return	6	1	length	4	
Call	7	1	charAt	5	
Return	8	1	charAt	5	
Call	9	None	isOperand	6	
Return	11	None	isOperand	6	
Call	12	2	push	7	
Return	13	2	push	7	
⋮					

call 頂点．メソッド呼出し位置に対応する頂点．

entry 頂点．メソッドの開始を示す頂点．

exit 頂点．メソッドの終了を示す頂点．

start 頂点．オブジェクトの生存期間の最初を示す頂点．

end 頂点．オブジェクトの生存期間の最後を示す頂点．

call 頂点は，Call イベントのメソッド呼出し位置 l の値ごとに存在する．同じメソッドを呼び出していても，その l の値が異なる命令であれば，それぞれに対応する別の call 頂点が存在する．entry 頂点は，メソッドの開始を表す頂点である．DOPG では，あるオブジェクトが持つメソッド中の，オブジェクトに対するメソッド呼出しの順序を，そのメソッドごとにグラフを持つことで表現する．そのメソッドごとのグラフの開始を表すのが entry 頂点となる．同様に，exit 頂点は，メソッドごとのグラフの終了を示す頂点である．start 頂点と end 頂点は，DOPG 全体での開始，終了をそれぞれが示す．

また，このグラフは以下のような種類の辺をもつ．

seq 辺．メソッド呼出しの順序を示す辺．

call 辺．メソッド呼出しを示す辺．

return 辺．メソッド呼出し元への復帰を示す辺．

seq 辺は call 頂点と任意の種類の頂点をつなぐ辺で，それぞれの頂点に対応するメソッド呼出しの順序を示す．例えば，call 頂点 A から call 頂点 B へ seq 辺は，call 頂点 A に対応する Call イベントの後に，call 頂点 B に対応する Call イベントが起こったということを表す．また，start 頂点から call 頂点 A への seq 辺は，call 頂点 A に対応する Call イベントが最初のイベントであることを表す．call 辺は，call 頂点と entry 頂点をつなぐ辺である．また，return 辺は，exit 頂点と call 頂点をつなぐ辺である．例えば，methodA の Call イベントに対応する call 頂点から出る call 辺は，methodA 内での動作を表すグラフの entry 頂点につながる．return 辺は，methodA 内での動作を表すグラフの exit 頂点から，methodA の Call イベントに対応する call 頂点への辺である．

実行履歴からこのグラフを作成する手順を以下に示す．ここで示す手順は，オブジェクトが単一のスレッドからのみメソッドを実行されている場合におけるものである．

1. start 頂点を生成し，これを頂点 $from$ とする．
2. 実行履歴から次の Call イベントを取得し，対応する call 頂点が既に存在すればそれを頂点 to とし，存在しなければ生成しそれを頂点 to とする．
3. 頂点 $from$ から頂点 to への seq 辺が存在しなければ，それを生成する．
4. 頂点 $from$ を，頂点 to とする，すなわち，$from$ に to を代入する．
5. 実行履歴にそれ以上 Call イベントがなければ 6 へ，そうでなければ 2 へ戻る．
6. end 頂点を生成し，頂点 $from$ からその end 頂点への seq 辺を生成する．

例として，図 1 のプログラムを，引数”1 2 +” で実行したときの実行履歴を用いる．表 2 は，実行履歴のうち，Stack クラスのオブジェクトに関するイベントをその発生時刻 t の順に並べたものである．つまり，o の値がすべて Stack オブジェクトを指している Call/Return イベントを集めたものである．命令位置を表す l は，図 1 中の行番号を用いて表記しているが，同じ行にある命令であっても，実際にはそれぞれで l の値は異なるため，この例では ”23.1”,”23.2” のように表記している．これを用いて，Stack オブジェクトに関するグラフを作成したものが図 2 である．call 頂点には，Call イベントが呼出したメソッドの名前と，その呼出し命令位置の ID（l）を記してある．

オブジェクトによっては，あるメソッドが呼出され，さらにそのメソッドの中で，メソッドを呼出されることがある．このような場合を示した実行履歴が表 3 である．Return イベ

ントはこのような場合を検出するために用いる．この場合，methodA 内でのオブジェクトの動作を表すグラフを作成する．start 頂点と end 頂点の代わりに，entry 頂点と exit 頂点をそれぞれ用い，上記と同様の手順で作成する．その後，methodA の Call イベントに対応する call 頂点から entry 頂点への call 辺を生成し，また，exit 頂点から methodA の call 頂点への return 辺を生成する．表 3 の実行履歴から作成されるグラフの一部を示したものが図 3 である．

また，マルチスレッドでオブジェクトが利用される場合について説明する．オブジェクトに対してのメソッド呼出しが複数のスレッドからされているような実行履歴は，各イベントの実行時刻 t の順に並べ，単一スレッドの実行履歴とみなし，グラフを作成する．表 4 は複数のスレッドからオブジェクトのメソッドを呼出している実行履歴の例である．この場合には，実行順に，methodA，methodB，methodC が，同一のスレッドから呼出されたものとしてグラフを同様の手順で作成することになる．一方，複数のスレッドから同時にメソッドが実行されているオブジェクトは，その動作を DOPG として蓄積することができない．あるオブジェクトについて，スレッドごとのコールスタックがあるとしたとき，任意の時刻において，２つ以上のコールスタックが空でない状態にならないことが，オブジェクトの動作を DOPG として抽出するための条件である．表 5 は，複数のスレッドから利用されるオブジェクトについての実行履歴の別の例である．この例では，methodB が呼び出された直後，スレッド 1 とスレッド 2 でコールスタックが空でなくなる．したがって，このような実行履歴は，単一スレッドのものとみなすことができないので，本手法で動作の抽出を行うことができず，抽出の対象外となる．

3.2.2 実行されたメソッドとその入力となったオブジェクト集合の取得

オブジェクトの動作に加え，実行されたメソッドの引数を記録する．これには，Entry イベントと，Parameter イベントを用いる．Entry イベントごとに，それに対応する Paramter イベントから，レシーバオブジェクトの ID である o および引数のオブジェクトの ID である P を取得する．入力が null である場合には，null をオブジェクト ID の代わりとする．数値などのオブジェクト以外の入力は考慮しない．結果として，メソッド m の第 n 引数に与えられたオブジェクト ID の集合 $O_{m,n}$ を得る．ただし，$n = 0$ はレシーバオブジェクトを表すものとする．例えば，表 6 のような実行履歴から $O_{m,n}$ を取得した結果は，表 7 のようになる．表中の methodB は，static なメソッドでレシーバオブジェクトがないものの例である．この場合，$O_{methodB,0}$ は空となる．

表 2: Stack オブジェクトに関する実行履歴

イベント	o	m	l
Call	1	Stack	3
Return	1	Stack	3
Call	1	push	7
Return	1	push	7
Call	1	push	7
Return	1	push	7
Call	1	pop	23.1
Return	1	pop	23.1
Call	1	pop	23.2
Return	1	pop	23.2
Call	1	push	23
Return	1	push	23

表 3: Call イベントが入れ子となっている実行履歴の例

イベント	m	l
⋮		
Call	methodA	1
Call	methodB	10
Return	methodB	10
Return	methodA	1
⋮		

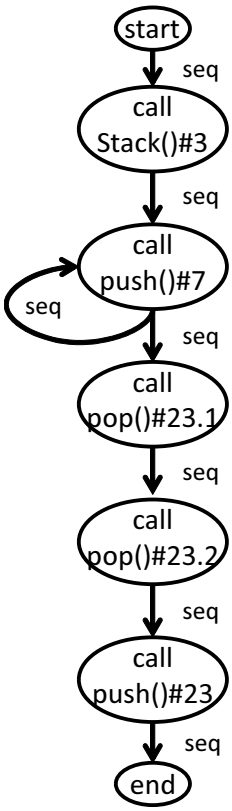

図 2: 表 2 の実行履歴から作成されるグラフ

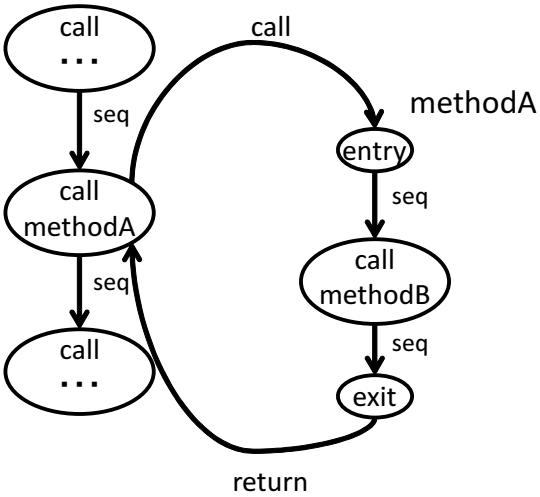

図 3: 表 3 の実行履歴から作成されるグラフ

表 4: 複数のスレッドから利用されるオブジェクトについての実行履歴の例

イベント	m	$thread$
Call	methodA	1
Return	methodA	1
Call	methodB	2
Return	methodB	2
Call	methodC	1
Return	methodC	1
⋮		

表 5: 本手法でグラフを作成することができない実行履歴の例

イベント	m	$thread$
Call	methodA	1
Call	methodB	2
Return	methodB	2
Return	methodA	1
⋮		

表 6: Entry イベントと Parameter イベントを含む実行履歴の例

イベント	o	m	P
⋮			
Entry	1	methodA	
Parameter	1	methodA	{2, 3}
⋮			
Entry	1	methodA	
Parameter	1	methodA	{null, 3}
⋮			
Entry	None	methodB	
Parameter	None	methodB	{1}
⋮			

表 7: 記録するメソッドとその入力となるオブジェクト集合の組

m	n	$O_{m,n}$
methodA	0	{1}
	1	{2, null}
	2	{3}
methodB	0	{}
	1	{1}

3.3 後の実行

プログラムの後の実行では，実行と同時に，出現しているオブジェクトの動作と事前の実行でのオブジェクトの動作を比較する．また，メソッドの実行が開始されるときに，そのメソッドの動作を実行履歴取得対象とするかどうかの判定を行う．本章では，オブジェクトの動作の比較の方法についてと，実行履歴取得対象の判定方法について説明する．

3.3.1 オブジェクトの動作の比較

オブジェクトの比較は，事前の実行で得られた DOPG と，本番環境での実行におけるオブジェクトの Call イベントと Return イベントを用いて行う．事前の実行で得られた DOPG をプッシュダウンオートマトンとみなし，初期状態を start 頂点とし，本番環境での実行におけるオブジェクトの Call イベントと Return イベントによって，オートマトンを状態遷移させる．オートマトンに遷移が定義されていないような入力が与えられたときにはオートマトンの状態を *sink* とし，これはオブジェクトとオートマトンの表す動作が一致していないことを表すとする．

実行中のプログラムに出現した 1 つのあるオブジェクト o があるとする．オブジェクト o は，動作の一致している DOPG 集合 D_o を持つ．オブジェクト o が出現した時点において，この D_o を，事前の実行で得られた DOPG のうち，オブジェクト o と同じクラスに属する DOPG の集合として初期化する．オブジェクト o に対する Call イベントまたは Return イベントが発生したとき，すべての $d \in D_o$ に対して，状態遷移処理を行う．そして，状態が *sink* となった DOPG d をすべて D_o から除外する．

オブジェクトと DOPG の，動作の比較のための状態遷移処理について説明する．ここでは，あるオブジェクト o と，ある DOPG d を比較するとする．比較は，DOPG の頂点の 1 つを現在の状態とし，その現在の状態を発生したイベントに応じて遷移させることで行う．まず，d の start 頂点を現在の状態とする．オブジェクト o に対して，Call イベントが発生

した場合，現在の状態の頂点から，Call イベントに対応する call 頂点への seq 辺が存在すれば，その call 頂点を現在の状態とする．このときに，上記のような seq 辺が存在しない場合，現在の状態を *sink* とする．また，その call 頂点から entry 頂点への call 辺が存在する場合には，その entry 頂点を現在の状態とする．最後に，Call イベントに対応する call 頂点をコールスタックにプッシュする．オブジェクト o に対して，Return イベントが発生した場合には，コールスタックから call 頂点を 1 つ取り出し，それを現在の状態とする．このときに，メソッドごとのグラフから復帰する場合には，exit 頂点への seq 辺が存在する場合にのみ上記の操作を行う．もし，exit 頂点が存在しない場合には，現在の状態を *sink* とする．

比較方法の例を示す．例では，あるオブジェクト o と動作の一致している DOPG の集合 D_o が，$D_o = \{d\}$ であるとする．このオブジェクト o と，DOPG d の比較する処理を説明する．オブジェクト o として，図 1 のプログラムに，入力として”1 2 + 3 +” を与えたときの実行で出現する Stack オブジェクトを用いる．o は，表 8 に示されるイベントを順に発生させられる．DOPG d は，図 2 のものであるとする．これは，図 1 のプログラムに，入力として”1 2 +” を与えたときに出現する Stack オブジェクトの動作を表した DOPG である．まず，DOPG d の start 頂点を現在の状態とみなし，コールスタックは空の状態にする（図 4）．最初のイベントは，$l = 3$ でのコンストラクタの Call イベントであるため，現在の状態である start 頂点から，seq 辺でつながれた，その Call イベントに対応する call 頂点へ現在の状態を更新し（図 5），コールスタックにこの call 頂点にプッシュする．この call 頂点は，call 辺を持っていないため，entry 頂点への状態遷移は生じない．次のイベントは，Return イベントなので，コールスタックから call 頂点をポップし，それを現在の状態とする．今の場合では，現在の状態は変化しない．次の Call イベントでも同様に，現在の状態が $l = 7$ の call 頂点へ遷移し，図 6 のようになる．以降のイベントに関しても同様に状態を遷移していくことで，$l = 23$ である push メソッドの Return イベントが生じたところで，図 7 のようになる．そしてその次の $l = 7$ である push メソッドの Call イベントでは，それに対応する call 頂点への seq 辺が，図 7 の状態の call 頂点からは存在していないため，この瞬間からオブジェクト o は DOPG d と異なる動作をしていると判断する．d は，o と動作が一致している DOPG の集合 D_o から取り除かれ $D_o = \emptyset$ となり，以後この d と o との比較処理は行われない．

3.3.2 実行履歴取得対象の判定

メソッドの実行が開始されたとき，そのメソッド内でのイベントを実行履歴取得対象とするかどうかを判定する．判定は，メソッドの入力に含まれるオブジェクトのうち，事前の実行でそのメソッドの入力となったすべてのオブジェクトと動作が不一致であるものが含まれるかどうかで行う．以下に判定方法を詳細に説明する．

表 8: 例として用いるオブジェクトに対するイベント列

イベント	m	l
Call	Stack	3
Return	Stack	3
Call	push	7
Return	push	7
Call	push	7
Return	push	7
Call	pop	23.1
Return	pop	23.1
Call	pop	23.2
Return	pop	23.2
Call	push	23
Return	push	23
Call	push	7
⋮		

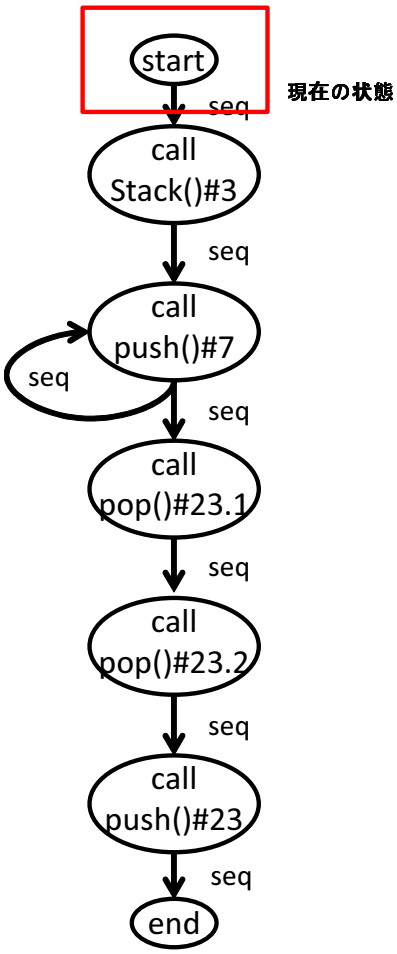

図 4: 初期状態の DOPG

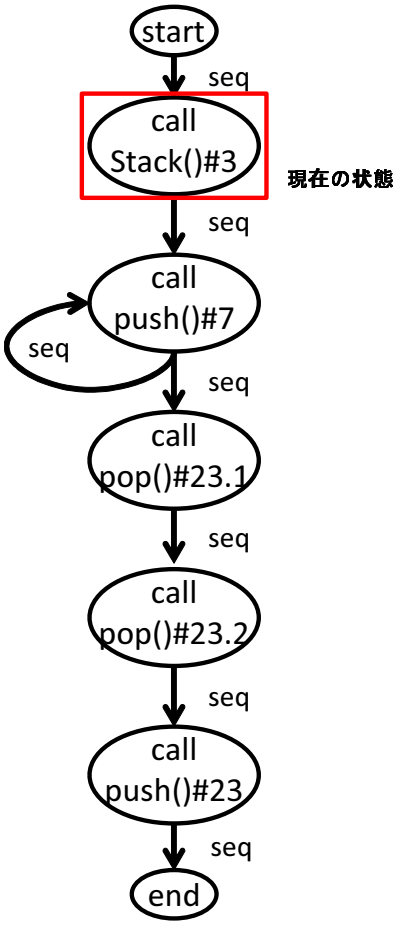

図 5: コンストラクタの呼出しイベントが発生した後の DOPG

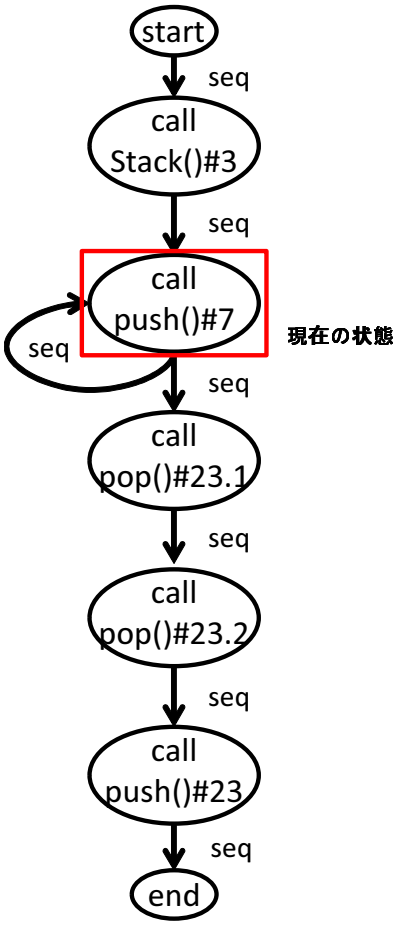

図 6: push メソッドの呼出しイベントが発生した後の DOPG

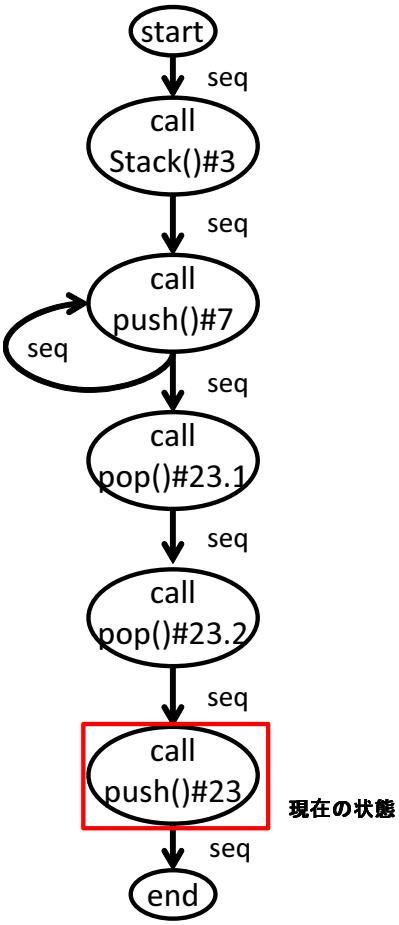

図 7: push メソッド呼出しから復帰イベントが発生した後の DOPG

メソッド m の実行が開始されたとき，メソッド m の第 n 引数に与えられたオブジェクト p_n と動作が一致している DOPG 集合を D_{p_n} とする．また，事前の実行で記録しておいた，メソッド m の第 n 引数に与えられたオブジェクト集合 $O_{m,n}$ に含まれるオブジェクトすべてについて，その動作を表す DOPG を事前の実行で得た記録から集め，その集合を D_{m_n} とする．このとき，メソッド m が実行履歴取得対象であるかどうかを表す $R(m)$ は，$R(m) = \exists n\{D_{p_n} \cap D_{m_n} = \emptyset\}$ と表される．n は $0 \leq n \leq$ メソッド M の引数の数，であり，$n = 0$ はレシーバオブジェクトを表すとしている．つまり，メソッドの入力となる各オブジェクトのうち，事前の実行でそのメソッドに入力として与えられたオブジェクトの動作とは一致しない動作をしているものが存在する場合に，そのメソッドが実行履歴取得対象となる．また，メソッドの引数が null である場合には，事前の実行でその引数に null が入力されているかを $O_{m,n}$ より調べ，されていないとき，実行履歴の取得対象とする．事前の実行で実行されていないメソッドについては，その入力のオブジェクトの動作に関わらず，無条件に実行履歴取得対象とする．

3.3.1 項の例を用いて，実行履歴取得対象の判定の例を示す．まず，図 1 のプログラムに数式”1 2 +” を与え，それを事前の実行とする．この実行では，1 つの Stack オブジェクトが出現し，その Stack オブジェクトの動作を表す DOPG が，図 2 である．calc メソッドの入力となったオブジェクトの DOPG 集合にそれが含まれることになる．つまり，その Stack オブジェクトの ID を 1 とすると，$D_{calc_1} = \{1\}$ となる．そして，同じプログラムに数式”1 2 + 3 +” を与えたときの実行を本番環境での実行であるとする．この実行でも 1 つの Stack オブジェクトが出現するので，それと動作が一致する DOPG 集合を持たせ，Call イベントと Return イベントが発生したときに比較を行う．3.3.1 項で示したように，$l = 23$ の push メソッドの Return イベントが発生するまで，この Stack オブジェクトは図 2 の DOPG と動作が一致しているとみなされるが，その次に $l = 7$ での push メソッド呼出しの Call イベントが発生したとき，動作が不一致であるとみなされる．この瞬間，この Stack オブジェクトと動作の一致している DOPG の集合は空となる．この後に calc メソッドの実行が開始されるとき，calc メソッドの第 1 引数となる Stack オブジェクトと動作の一致している DOPG の集合 D_{p_1} は空となるため，$R(calc)$ は真となり，この calc メソッドの実行は実行履歴取得の対象であると判定される．一方，Stack オブジェクトが DOPG と動作が一致しているとみなされている間の calc メソッドの実行は，実行履歴取得対象にならない．

4　評価実験

本研究では，提案手法によりどの程度実行履歴の記録量が削減できるかを調査するため，評価実験を行った．評価実験は，DaCapo ベンチマーク [4] に含まれる 4 つの Java アプリケーションを対象に行った．DaCapo ベンチマークは，様々なアプリケーションを実行できるベンチマークソフトである．実行時の引数により，実行するアプリケーションを選択できる．また，アプリケーションの実行の規模も，small，default，large から選択することができる．実行の規模とは，たとえば画像処理を行うアプリケーションである batik では，処理を行う画像の枚数で表される．実験の対象としたアプリケーションは，DaCapo ベンチマークに含まれるアプリケーションのうち，batik，fop，luindex，pmd の 4 つである．

4.1　実験方法

提案手法では，プログラムの実行履歴を取得する必要がある．本研究では，表 9 のイベント情報を含む実行履歴を使用し，評価実験を行った．この実行履歴は，それを用いてプログラムの実行を再現することができるようなイベント情報を含んでいる．

評価実験は，DaCapo ベンチマークの，small 規模での実行をプログラムの事前の実行として用い，small よりも規模の大きい default の実行を本番環境での実行として用いる．これは，事前の簡易な実行より，本番環境での実行の方がその規模は大きいという想定に基づいている．small 規模での実行で，DOPG とメソッドの入力となったオブジェクト集合の抽出を行う．default 規模の実行では，各オブジェクトに対して，DOPG との動作の比較を行い，各メソッド実行が実行履歴取得対象となるかの判定を行う．

また，各メソッド実行の実行パスを記録する．実行パスは，メソッド実行中に通過したバイトコード上のラベル集合として求める．つまり分岐命令の結果が異なる場合には，異なるパスとして扱う．一方で，ループ文の繰り返し回数の違いは，実行パスの違いには含めない．評価実験では，small 実行での実行パスをメソッドごとにまとめ，default 規模の実行での，各メソッド実行のパスと比較する．small 規模の実行には存在しない default 規模の実行の実行パスをメソッドの未知の振舞いとする．

4.2　制限

評価実験では，以下のオブジェクトを，DOPG 抽出の対象から除外した．

- 実行履歴にメソッド呼出しイベントが 1 つも含まれないオブジェクト
- 複数のスレッドから同時にメソッドを呼出されているオブジェクト
- String クラスのインスタンスであるオブジェクト

表 9: 評価実験で利用した実行履歴に含まれるイベント

イベント種類	意味
Call	メソッド呼出し
Entry/Exit	メソッド実行の開始および終了
Parameter	Call/Entry イベントの引数
Return	メソッド呼出しからの復帰
Field Read/Write	フィールドの読込みおよび書込み
Array Read/Write	配列への読込みおよび書込み
Label	ラベルへのジャンプ命令
New	配列・オブジェクトの生成
Object Initialized	オブジェクトの初期化
Throw	例外のスロー
Catch	例外のキャッチ
InstanceOf	instanceof 演算の実行
Monitor Enter/Exit	排他制御の開始および終了

String 型は，メソッド呼出しによって内部の状態が変化せず，メソッドの呼出し順序を比較する意味が少ないうえに，プログラム実行中に生成と消滅を繰り返し，大量のオブジェクトが出現する．よって提案手法を Java プログラムに適用するにあたり，String 型のオブジェクトは手法の対象外とした．

4.3 評価尺度

評価実験で用いる評価尺度を以下のように定義する．

R 実行履歴の取得対象となるイベントの割合

UB 未知の振舞いをするメソッド実行の割合

UBR 実行履歴の取得対象となるメソッド実行のうち，未知の振舞いをするメソッド実行の割合

RUB 未知の振舞いをするメソッド実行のうち，実行履歴の取得対象となるメソッド実行の割合

UBR' small 規模の実行では実行されなかったメソッドを除外したときの UBR 値

RUB' small 規模の実行では実行されなかったメソッドを除外したときの RUB 値

提案手法では，small 規模では実行されなく default 規模の実行では実行されるようなメソッドが，必ず実行履歴の取得対象となる．また，このようなメソッド実行は，必ず未知の振舞いをしているとみなされる．このようなメソッド実行の数によっては，UBR と RBU の値が大きく影響を受ける．そこで UBR' と RBU' は，そのようなメソッドの影響を受けないように提案手法を評価するために用いる．

4.4 実験結果

評価実験の結果について説明する．表 10 と表 11 は各アプリケーションの small 規模での実行で抽出した DOPG の情報を示したものである．表 10 では，DOPG の数と，DOPG を抽出したクラスの数，そして実験で無視された String 型オブジェクトの数を示している．1 度もメソッドを呼出されず，DOPG 作成が不可能であるオブジェクトの数は，この表には含まれていない．なお，複数のスレッドからメソッドを同時に呼び出されているオブジェクトは，pmd において 1 つ存在し，他のアプリケーションでは存在しなかった．表 11 は，その DOPG ファイルサイズの分布を示したものである．

評価実験の結果を表 12 に示す．実験結果の R の値から，提案手法によって，23.7%から 62.4%の割合のメソッド実行が，実行履歴取得の対象となったことがわかる．UB の値は各アプリケーションの未知の振舞いをするメソッド実行の割合を示しているので，未知の振舞いをするメソッド実行のみを，実行履歴取得対象として選べた場合には，R の値と UB の値が等しくなる．しかし，実験結果から，R の値は UB の値よりも大きくなっている．これは，small 規模での実行と同じ動作をしているメソッド実行も，実行履歴の取得対象としてしまっていることを示している．それを示すように，UBR の値は非常に小さくなっていることが確認できる．表 12 では省略しているが，UBR' の値はすべてのアプリケーションについて，ほぼ 0%であった．このような余分に実行履歴を取得してしまうようなメソッド実行に対しては，あきらかに実行パスが，事前実行と異ならないようなメソッドを，あらかじめ実行履歴から外しておくことで，対応できるものもあると考えられる．例えば，ただフィールドの値を取得したり，更新したりするだけの，getter，setter が挙げられる．

RUB の値は，未知の振舞いをするメソッド実行のうち，提案手法によって実行履歴を取得できたメソッド実行の割合を示している．small 規模では実行されないメソッドは，必ず実行履歴取得対象とすることができるので，この値はすべてのアプリケーションで大きくなっている．そのようなメソッドの影響をなくして計算した RUB' の値は，luindex と pmd では大きくなっていて，batik と fop では小さくなっている．これは batik と fop で，未知の振舞いをするメソッド実行を，実行履歴取得対象にできていない，取りこぼしが多いこ

表 10: small 規模の実行のイベント数および DOPG 数

	イベント数	DOPG 数	クラス数	String 型の DOPG 数
batik	168,534,669	171,138	656	4,812
fop	74,328,761	208,113	746	23,141
luindex	73,106,032	15,295	174	414
pmd	17,303,153	27,382	415	5,702

表 11: DOPG のサイズ分布 (Byte)

	最小値	中央値	平均値	最大値
batik	202	738	1,517	32,101
fop	112	723	1,461	73,946
luindex	202	578	591.4	28,723
pmd	202	615	1,509	160,443

とを示している．提案手法では，入力にオブジェクトを含まないようなメソッドの実行を，実行履歴取得対象に含めることができない．例えば，fop においては，このような取りこぼしたメソッド実行の多くが，`org.apache.fop.util.CharUtilities` クラスのメソッドで，isAdjustableSpace メソッド，classOf メソッド，isAnySpace メソッド，isNonBreakableSpace メソッドの実行であった．これらのメソッドはすべてクラスメソッドで，引数にもオブジェクトを取らない．このように，int 型や char 型等のプリミティブな引数の値によって，メソッドの振舞いに直接影響を受けるようなメソッド実行は，提案手法では対応できないことを示している．このようなプリミティブな値の違いによって生じる実行パスは，単体テストで事前に網羅しやすいものと考えられる．

表 13 は，実験で用いた実行履歴に含まれるイベントのうち，実行履歴取得対象となったイベントの数を示している．これは，あるメソッドが実行履歴取得対象であるとされたときに，そのメソッドの Entry イベントから Exit までのイベントを記録するとして計算した．取得対象でないとされたときは Entry イベントから Exit イベントまでのイベントを，Entry と Exit イベントを含めて，記録対象から除外した．実行履歴中の各イベントが，すべて同じサイズであると仮定すれば，表中の記録割合が提案手法適用後の実行履歴のサイズを示している．

表 12: 評価実験の結果 （単位：%）

	batik	fop	luindex	pmd
R	26.3	23.7	58.6	62.4
UB	22.2	2.2	8.2	2.5
UBR	84.4	8.2	13.9	4.0
RUB	99.9	87.3	99.9	98.1
RUB'	18.5	5.1	97.1	92.2

表 13: 実行履歴削減量

	batik	fop	luindex	pmd
総イベント数	712,010,905	417,587,236	2,160,725,683	1,425,282,440
記録イベント数	227,794,693	95,211,219	1,025,008,169	958,311,316
記録割合	31.99%	22.80%	47.43%	67.23%

5　まとめ

ソフトウェアがリリースされた後に生じる障害を，開発環境で再現するために，実行履歴を記録する必要があるが，詳細な動作の情報を含む実行履歴はその量が非常に大きい．そこで，事前にプログラムを実行しておき，そのときの動作とは違う動作をする部分のみについて実行履歴を記録することで，実行履歴の量を削減する手法を提案した．評価実験では，実行履歴の量が 22.8%から 67.23%の量に減らすことができることを確認した．また，2 つのアプリケーションで，事前の実行では確認されていない動作をするメソッドの実行を，実行履歴の記録対象とすることができていることを確認した．

本研究の今後の課題としては，より多くの種類のアプリケーションに対して，提案手法を適用し，どのようなアプリケーションや，またはメソッドに対して，提案手法が効果的できるのかを詳細に調査することがあげられる．また，それの結果から，余分な記録や，記録漏れへの対応を考える必要があるとも考えている．さらに，プログラムの実行と同時に本手法を適用したときに生じるオーバーヘッドを計測し，手法のスケーラビリティについて考察する必要があることも挙げられる．

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石尾隆助教に深く感謝いたします.

本研究において，終始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および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] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE 2013, pp. 362–371, 2013.

[2] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE 2009, pp. 34–44, 2009.

[3] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE 2007, pp. 261–270, 2007.

[4] DaCapo. http://www.dacapobench.org/.

[5] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2009, pp. 550–554, 2009.

[6] FindBugs. http://findbugs.sourceforge.net/.

[7] gzip. http://www.gzip.org/.

[8] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proceedings of the 18th International Workshop on Program Comprehension*, IWPC2002, pp. 159–168. IEEE, 2002.

[9] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. Ocat: Object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA 2010, pp. 159–170, 2010.

[10] Shrinivas Joshi and Alessandro Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, ICSM2007, pp. 234–243, 2007.

[11] JUnit. http://junit.org/.

[12] Bil Lewis. Debugging backwards in time. In *Proceedings of the 5th International Workshop on Automated Debugging*, AADEBUG 2003, pp. 225–235, 2003.

[13] Jan Lönnberg, Mordechai Ben-Ari, and Lauri Malmi. Java replay for dependence-based debugging. In *Proceedings of the 9th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2011, pp. 15–25. ACM, 2011.

[14] Salman Mirghasemi, John J Barton, and Claude Petitpierre. Querypoint: moving backwards on wrong values in the buggy execution. In *Proceedings of the 8th Joint Meeting of the 13th European Software Engineering Conference and 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ESEC/FSE 2011, pp. 436–439, 2011.

[15] 宗像聡, 石尾隆, 井上克郎. 類似した振舞いのオブジェクトのグループ化によるクラス動作シナリオの可視化. 情報処理学会研究報告，2009-SE-163，Vol.2009，No.31, pp. 225–232, 2009.

[16] Joehen. Quante and Rainer. Koschke. Dynamic object process graphs. *The Journal of Systems and Software*, Vol. 81, No.4, pp. 481–501, 2008.

[17] Robin Salkeld and Gregor Kiczales. Interacting with dead objects. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages & Applications*, OOPSLA 2013, pp. 203–216. ACM, 2013.

[18] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezheng Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. Language-based replay via data flow cut. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2010, pp. 197–206, 2010.