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と評価

脇阪 大輝[1,a)]　石尾 隆[1,b)]　井上 克郎[1,c)]

概要：ソフトウェアは開発期間中にテストを実施されるが，テストによって発見されなかった欠陥はソフトウェア利用者のもとで障害を引き起こす．障害を開発環境で再現する場合，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が，詳細な実行履歴のデータ量は膨大なものとなってしまう．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，オブジェクトの動作を Dynamic Object Process Graph（DOPG）として抽出しておく．利用者の環境での実行では，DOPG として記録された動作と一致しない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．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，実行履歴の量を 22.8%から 67.23%に減らせることを確認した．

キーワード：動的解析，実行履歴，障害再現

1. はじめに

ソフトウェアに含まれる欠陥を発見，除去するために，開発期間中には静的解析や動的解析の技術を利用したテストが実施される．しかし，すべての欠陥が，事前のテストによって発見されるとは限らない．完全なテストは難しく，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], [8]．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．

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 [15]．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．

リリース後に発生する障害を開発環境で迅速に再現するために，本番環境でのプログラムの実行履歴を記録しておけばよい [9]．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．プログラムのすべて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

1　大阪大学大学院情報科学研究科
a)　h-wakisk@ist.osaka-u.ac.jp
b)　ishio@ist.osaka-u.ac.jp
c)　inoue@ist.osaka-u.ac.jp

© 1959 Information Processing Society of Japan

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，実行履歴の量を約 22.8%から 67.23%に減らすことができた．また，そのうちの 2 つでは，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 割を実行履歴の記録対象とすることができた．

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．3 章で提案手法である実行履歴削減手法を説明し，4 章では実施した評価実験について説明する．最後に 5 章ではまとめと今後の課題について述べる．

2. 背景

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，プログラムの実行中に発生したイベントが時系列順に並んでいる．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[10] 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，任意の時点の状態を再現する手法である．このようなデバッグ手法を実現するためには，プログラムの動作の詳細，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実行履歴が必要である．並列プログラムのデバッグにおいては，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 [11]，そうではないプログラムのデバッグにおいては有効であるとされている [12]．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．

利用者のもとで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，プログラムの実行時性能に悪影響を及ぼす．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．このような性能への悪影響を避けるために，実行履歴の量を少量にとどめる方法が必要である．

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], [16]．これらの手法は，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再実行することで，プログラム利用者の環境での実行を再現する．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おき，次に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で置き換える．これにより，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，少ない実行履歴でプログラムの実行を再現できる．しかし，このような手法は，プログラムを最初から実行するという手法の性質上，実行時間が非常に長いサーバーアプリケーション等の実行を再現するには不向きである．本研究では，メソッド実行の単位で実行履歴を記録することで，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]．この手法では，実行履歴を記録する範囲を小さく抑えることができる一方で，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．また，欠陥を含んでいそうな箇所をソースコードの位置として推定しても，その箇所が何度も実行される場合，すべてが障害に関係するとは限らない．提案手法では，同じメソッドでも，障害に関係しそうにない実行については，実行履歴を記録しないことが可能である．

実行履歴に対してファイル圧縮技術 [6] や，実行履歴の圧縮技術 [7] 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ている．本研究で得られる実行履歴にも，これらの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は，実行履歴の比較を行っている以下の事例に基づいている．

宗像らの研究 [13] 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．本研究では，オブジェクトの動作が同一であれば，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として記録する．

3. 提案手法

本研究では，Java プログラムを対象とし，記録する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，事前の実行としてプログラムの簡易な実行を行い，出現したオブジェクトの動作を抽出し，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．後の実行では，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し，メソッド実行開始時に実行履歴取得の対象かどうかを判定することで，事前の実行に含まれない動作の詳細な実行履歴を記録する．

3.1 提案手法で用いる実行履歴

提案手法で用いる実行履歴は，少なくとも以下のイベントを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 $\langle t, thread, o, m, P\rangle$.　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．

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 は呼び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．例として，図 1 に示すプログラムは．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．なお，表中では，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 イベントの引数情報を用いる．

```
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: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

表 1　実行履歴の例

イベント	t	o	m	l	P
Entry	1	None	main		{0}
Call	2	2	Stack	3	
Return	3	2	Stack	3	
Call	4	1	length	4	
Return	5	1	length	4	
Call	6	1	charAt	5	
Return	7	1	charAt	5	
Call	8	None	isOperand	6	
Return	9	None	isOperand	6	
Call	10	2	push	7	
Return	11	2	push	7	
⋮					

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）[14] 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．

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，DOPG に対応するオブジェクトのメソッドを

直接呼び出す，あるいはそのようなメソッドを間接的に呼び出す 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，l の値が異なれば，それぞれ独立した頂点となる．entry 頂点は，Call イベントに対応して実行されたメソッドの開始を表す頂点である．exit 頂点は，そのメソッドの終了を表す．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．

DOPG 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．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．

オブジェクトが単一のスレッドからのみ使用されている場合，DOPG は定義 [14] に従って作成する．図 2 は，図 1 のプログラムを引数 “1 2 +″ で実行したときの，Stack オブジェクトについての DOPG を示したものである．

マルチスレッドでオブジェクトが使用される場合は，オブジェクトに対するイベントをタイムスタンプ t の値の順に並べ，単一スレッドの実行履歴とみなして DOPG を作成する．具体的には，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，任意の時点において，高々 1 つのスレッドのコールスタックにしか要素が存在しないとき，それらの呼出しがシングルスレッドで時系列順に発生したとみなす．提案手法では，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については，その動作を DOPG として抽出することができない．本手法では，そのようなオブジェクトを動作の抽出の対象外とした．

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 イベントを用い，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無視する．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 2 のような Entry イベントのみを取り出

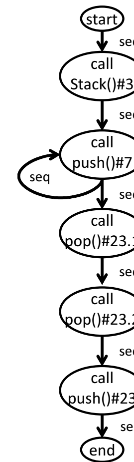

図 2 図 1 のプログラムを引数 “1 2 +″ で実行したときの Stack オブジェクトについての DOPG

表 2 Entry イベントの系列の例

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

表 3 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}

した実行履歴から $O_{m,n}$ を取得した結果は，表 3 のようになる．表中の methodB はクラスメソッドで，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．

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

オブジェクトの動作の比較は，事前の実行で得られた DOPG と，後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，後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

© 1959 Information Processing Society of Japan

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．

実行中のプログラムに出現したオブジェクト o に対して，動作の一致している DOPG 集合 D_o を対応付ける．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．

あるオブジェクト o と DOPG d の，動作の比較のための状態遷移処理は，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 頂点を初期状態とする．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．そのような 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，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，もし d に Return に対応する 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 o を，図 2 に示す DOPG d と比較する．o は，表 4 に示されるイベントを順に発生させられる．まず，DOPG d の start 頂点を現在の状態とみなし，スタックは空の状態にする（図 3（a））．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 3（b）），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，スタックから call 頂点を取り出し，それを現在の状態とする．この例では，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 頂点へ遷移する．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 3（c）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 3（c）の状態の call 頂点からは存在していないため，状態は $sink$ となる．この瞬間からオブジェクト o は，DOPG d の表す動作と一致していない動作をしていると判定される．d は，D_o から取り除かれ，$D_o = \emptyset$ となる．

表 4 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
⋮		

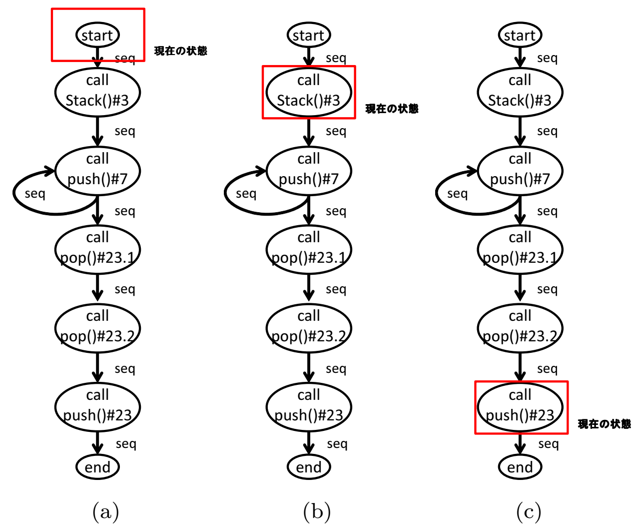

図 3 DOPG の操作例

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どのオブジェクトとも動作が一致しない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．

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の動作

© 1959 Information Processing Society of Japan

を表す DOPG 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．つまり，メソッドの入力となる各オブジェクトに，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どの動作とも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，null が一度も入力されていないとき，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 2 とすると，$D_{calc_1} = \{2\}$ 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後の実行とする．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，提案手法の以下の特性を評価するために実験を行った．

実行履歴の削減能力 実行履歴の量をどの程度削減することが可能であるか.

未知の振舞いの記録能力 事前の実行では観測されていないプログラムの動作に限定して実行履歴を記録できるか.

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 5 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，small 実行の実行経路をメソッドごとにまとめ，default 規模の実行の各メソッド実行の実行経路と比較する．各メソッド実行の実行パスを記録する．実行経路は，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，異なる実行経路として扱う．一方で，ループ文の繰り返し回数の違いは，実行経路の違いに含めない．評価実験では，small 規模の実行には存在しない実行経路をたどる，default 規模の実行のメソッド実行を**未知の振舞いをするメソッド実行**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

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

表 5 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

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メソッド実行の割合

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

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メソッド実行とみなされる．このようなメソッド実行の数によっては，RUB の値が大きく影響を受けるため，そのようなメソッドの影響を受けないように RUB' の値を用いる．

4.4 実験結果および考察

評価実験の結果について説明する．表 6 では，small 規模の実行のイベント数と，抽出した DOPG の数，そして実験で無視した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．

評価実験の結果を表 7 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の取得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，本手法によって，未知の振舞いをするメソッド実行のみを実行履歴の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，実験結果では，R の値が 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を，実行履歴の取得対象としてしまっていることを表す．それを示すように，UBR の値は小さくなっている．small 規模では実行されないメソッドを取り除いた UBR の値はすべてのアプリケーションで約 0%となった．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，あきらかに実行経路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の取得対象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および更新を行うだけの，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，必ず実行履歴の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る．これは，未知の振舞いをするメソッド実行の多くを，提案手法により実行履歴の記録対象とできていることを示している．一方，batik と fop ではこの値が小さくなっている．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ず，多くを取りこぼしていることを示している．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．このように提案手法は，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メソッドには対応できない．このようなプリミティブな型の値の違いによって生じる実行経路は，単体テストで事前に網羅しやすいものと考えられる．したがって本研究では，このようなメソッドの実行履歴を記録する必要性は少ないとしている．

表 8 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

表 6 small 規模の実行のイベント数および DOPG 数

	イベント数	DOPG 数	対象外オブジェクト数
batik	168,534,669	171,138	4,812
fop	74,328,761	208,113	23,141
luindex	73,106,032	15,295	414
pmd	17,303,153	27,382	5,703

表 7 評価実験の結果 (単位：%)

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

表 8 実行履歴削減量

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

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．luindex や pmd では，他と比較して記録割合の値が大きくなっている．これらのアプリケーションは，default 規模の実行に比べ，small 規模の実行のイベント数が非常に少ない．したがって少ないイベントを記録することで，多くのイベントを記録を減らしたという点で，提案手法が有効であったと考えることができる．事前の実行がより豊富であれば，より多くの記録を削減することができる可能性もある．

5. まとめと今後の課題

プログラム利用者のもとで発生する障害を開発環境で再現するためには，実行履歴が有用であるが，詳細な動作の情報を含む実行履歴はデータ量が非常に大きい．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一致しない動作をしそうなメソッド実行についてのみ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の多くを，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．

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またはメソッドに対して，提案手法が効果的であるのかを詳細に調査することが考えられる．また，その結果から，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，手法の実用性について考察する必要がある．

謝辞 本研究は，科研費 課題番号:25220003 および科研費 課題番号:23680001 の助成を得た．

参考文献

[1] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proc. of ICSE*, pp. 362–371, 2013.
[2] T M. Chilimbi, B. Liblit, K. Mehra, A. V Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proc. of ICSE*, pp. 34–44, 2009.
[3] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proc. of ICSE*, pp. 261–270, 2007.
[4] DaCapo. http://www.dacapobench.org/.
[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proc. of ASE*, pp. 550–554, 2009.
[6] gzip. http://www.gzip.org/.
[7] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proc. of IWPC*, pp. 159–168, 2002.
[8] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: Object capture-based automated testing. In *Proc. of ISSTA*, pp. 159–170, 2010.
[9] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *Proc. of ICSM*, pp. 234–243, 2007.
[10] Bil Lewis. Debugging backwards in time. In *Proc. of IWAD*, pp. 225–235, 2003.
[11] Jan Lönnberg, Mordechai Ben-Ari, and Lauri Malmi. Java replay for dependence-based debugging. In *Proc. of PADTAD*, pp. 15–25, 2011.
[12] S. Mirghasemi, J. J Barton, and C. Petitpierre. Query-point: moving backwards on wrong values in the buggy execution. In *Proc. of ESEC/FSE*, pp. 436–439, 2011.
[13] 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.
[14] J. Quante and R. Koschke. Dynamic object process graphs. *Journal of Systems and Software*, Vol. 81, No.4, pp. 481–501, 2008.
[15] R. Salkeld and G. Kiczales. Interacting with dead objects. In *Proc. of OOPSLA*, pp. 203–216. ACM, 2013.
[16] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *Proc. of FSE*, pp. 197–206, 2010.

© 1959 Information Processing Society of Japan