特別研究報告

題目

実行履歴の照合によるオブジェクトの振舞い予測手法の提案

指導教員
井上 克郎 教授

報告者
脇阪 大輝

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

実行履歴の照合によるオブジェクトの振舞い予測手法の提案

脇阪 大輝

内容梗概

オブジェクト指向プログラムは，多数のオブジェクトの相互作用によって動作する．そのため，プログラムの状態はオブジェクトの動作で決定される．興味のあるプログラムの動作に対応するオブジェクトを詳しく調査するためには，プログラムの実行中に，分析したいオブジェクトを興味ある動作を行うよりも前に特定しなくてはならない．しかし，オブジェクト指向プログラムが実行される際，多数のオブジェクトが生成されるため，オブジェクトを特定するのは容易ではない．

そこで，本研究では興味のあるプログラムの動作に関係するオブジェクトを特定するため，１度実行したプログラムのオブジェクトの振舞いに基づき，オブジェクトの振舞いを予測，提示する手法を提案する．本手法では，オブジェクトの振舞いの事例をプログラムの実行履歴から収集し，Dynamic Object Process Graph (DOPG) という中間表現を経由して，オートマトンとして抽出する．次に，もう一度プログラムを実行するときに，各オブジェクトに対してオートマトンを動作させ振舞いの一致する事例を探す．

また，提案手法により，オブジェクトの振舞いが予測可能であるかを確認するため，評価実験を行った．実験では，DaCapo ベンチマークに含まれるアプリケーションを実行させた際に得られる実行履歴を用いて，提案手法により各クラスのオブジェクトの振舞いを予測できるかを調べた．実験結果より，実験対象に含まれるクラスのうち，34%のクラスが振舞いの予測が必要であり，その中の 70%について提案手法により振舞いを予測することが可能であるとわかった．

主な用語

オブジェクト指向プログラム
プログラム理解
実行履歴
振舞い予測
オートマトン

目 次

1　まえがき

オブジェクト指向プログラムは，多数のオブジェクトの相互作用によって動作する．オブジェクトとは，実行時にメモリ上に生成され，メソッド呼び出しによって要求された処理を実行する機能部品である．プログラムの動作を理解するには，オブジェクトの振舞いを理解することが重要となる [3].

オブジェクトの機能は，ソースコード上ではクラス単位で記述されるが，１つのクラスのオブジェクトは，一般に複数生成される．同じクラスのオブジェクトであっても，使用される状況や，オブジェクト自身の状態によって，プログラム内部でのオブジェクトの振舞いは異なる場合がある [9]．ソースコードだけから，そのようなオブジェクトの区別を正確に行うことはできないため [7, 13]，プログラムを実際に実行し，オブジェクトの振舞いを調査する必要がある．

開発者が，プログラムのある特定の動作に対応するオブジェクトの振舞いを調査する場合，まず，実行中のプログラムから，その動作に対応するオブジェクトを特定しなくてはならない．対話的デバッガを用いてこの作業を行う場合，注目している動作に対応する命令が実行されるよりも前にブレークポイントを配置してプログラムの実行を一時停止し，注目しているオブジェクトが処理を実行しようとしているかどうかをステップ実行によって確認していくことになる．ブレークポイントによって一時停止を起こしたオブジェクトが，実際に開発者が求める振舞いのオブジェクトかどうかは，その振舞いを観測してみるまでは分からないため，同一クラスではあるが無関係なオブジェクトについてもステップ実行を逐一行っていく作業は，負担の大きい作業となっている．

そこで，本研究では，開発者が調査すべきオブジェクトを特定するために，プログラムの実行中に，各オブジェクトがこれから行うであろう振舞いを予測し，可視化することを考える．宗像ら [9] の研究では，オブジェクトが多数存在する場合でも，その呼び出し関係などを比較することで，オブジェクトの振舞いを少数のグループに分類することが可能であることが示されている．プログラムを１度実行しておき，オブジェクトの振舞いを事前にグループ化しておくことで，次のプログラムの実行において，新しいオブジェクトの振舞いがどのグループに該当するかを調査することで，オブジェクトの振舞いを特定する．ここでは，プログラムを２度実行したとき，それがほぼ同じ振舞いを行うことを仮定するが，これは，デバッグやプログラム理解の作業においては，ある特定の入力に対応した振舞いを分析する作業が重要であるためである．

本研究では，1 度実行したプログラムのオブジェクトの振舞いに基づいて，オブジェクトの振舞いを予測し提示する手法を提案する．本手法では，オブジェクトの振舞いとして，オブジェクトに対するメソッド呼び出し列の事例をプログラムの実行履歴から収集する．具体

的には，オブジェクトごとの実行履歴を Dynamic Object Process Graph (DOPG) [12] という表現に変換し，これをもとに，オブジェクトごとの振舞いを表現したオートマトンを抽出する．この結果，各クラスに対して，そのクラスのオブジェクト群の振舞いに対応したオートマトンの集合が得られる．次にプログラムを実行するときには，ある 1 つのオブジェクトに対するメソッド呼び出し系列をすべてのオートマトンに入力として与え，系列を受理しないオートマトンを順番に取り除いていき，ただ 1 つ残ったオートマトンを，そのメソッド呼び出し系列に対応する事例として認識する．オートマトンには，入力されたメソッド呼び出し系列の続きとして受理しうるメソッド呼び出しの系列が記録されており，それを，そのオブジェクトが今後受け取るメソッド呼び出しの系列であるとみなす．

本研究において，オートマトンの抽出に重要な役割を果たしているのが DOPG である．DOPG は，元々はプログラム理解支援を目的として提案された実行履歴の表現であり，ソースコード上での制御の流れを反映するという特徴がある [11]．ある 1 つのオブジェクトが受け取ったメソッド呼び出し系列を受理するオートマトンは，自由に作成することができるが，DOPG の表現はただ 1 つに定まる．DOPG をオートマトンに変換するルールを定義することで，ソースコード上での振舞いに対応したオートマトンを抽出している．

本手法が，実際にオブジェクトの振舞いを予測可能であるかを確認するため，評価実験を行った．本実験では，DaCapo ベンチマークの一部を実行し，そのうち 803 個のクラスについて，オートマトン集合を抽出した．その結果，約 66 %のクラスについては，そのクラスのオブジェクトに対応するオートマトンがただ 1 つとなることが示された．残る 34%のクラスでは，複数のオートマトンが存在しており，オブジェクトの振舞いの事例から，実際の振舞いを特定する必要があった．その中の 70 %（全体の 24 %）については，提案手法によって，オブジェクトの振舞いを予測することが可能であることを確認した．また，予測できるのは，平均で 4 つのメソッド呼び出しであることを示した．

本論文の構成は次の通りである．まず，2 章で研究の背景としてオブジェクトの振舞いを表現する Dynamic Object Process Graph について述べ，3 章でオブジェクトの振舞いを予測する手法の詳細を説明し，4 章で評価実験について述べる．5 章で関連研究について述べ，最後に 6 章でまとめと今後の課題について述べる．

2 Dynamic Object Process Graph

プログラムのオブジェクトの振舞いを表現する既存手法として，Dynamic Object Process Graph (DOPG) というグラフがある．DOPG とは実行履歴から動的に生成され，あるオブジェクトが受取った生成命令やメソッド呼出し命令のイベントの列を有向グラフで表現したものである．プログラム中のオブジェクト 1 つにつき 1 つの DOPG が生成されるが，複数のスレッドから使用されるオブジェクトについては，スレッドの数だけ DOPG が生成される．DOPG は，オブジェクトの持つメソッド内での振舞いと，それ以外での振舞いを表すグラフをつないで作成される．

DOPG のノードには以下のものがある．

start ノード プログラムの開始を表す．DOPG は常にこのノードから始まる．

end ノード プログラムの終了を表す．DOPG は常にこのノードで終わる．

create ノード オブジェクトの生成を表す．生成されたソースコードでの位置を含む．DOPG は必ず start ノードのあとにこのノードがある．

call ノード メソッド呼出しを表す．呼び出すメソッドの名前と呼出しを行うソースコードでの位置を含む．

entry ノード メソッド処理の開始を表す．メソッド内での振舞いを表すグラフは必ずこのノードから始まる．

return ノード メソッド処理の終了を表す．メソッド内での振舞いを表すグラフは必ずこのノードで終わる．

DOPG のエッジには以下のものがある．

call エッジ メソッド呼出しを表す．call ノードから，その call ノードの呼出しメソッドのグラフの entry ノードへひかれる．

return エッジ メソッド呼出しからの復帰を表す．メソッドのグラフの return ノードから，呼出し元の call ノードへひかれる．

seq エッジ イベントの順序を表す．A から B への向きでひかれた場合，A の次に B が行われたことを意味する．

DOPG は start ノードから始まり，その次は create ノードが続く．プログラム実行中に起こったメソッド呼出しの順に call ノードが seq エッジでつながれ，最後に end ノードで終わ

る．メソッドごとのグラフの entry ノードが呼出し元の call ノードと call エッジでつながれる．また，return ノードが呼出し元の call ノードと return エッジでつながれる．DOPG の各 create ノードと call ノードはイベントに対応する命令のソースコード上の位置に対応しており，ソースコード上におけるループ構造を抽象化する．具体的には，既に同じイベントに対応するノードが DOPG に含まれる場合に，新しいノードを追加せず，既にあるノードへのエッジをひく．これにより，実行時のループ回数によってグラフが変化することがなくなる．

複数のスレッドから使用されるオブジェクトについての DOPG は，スレッドごとにグラフが生成される．この場合，それぞれのグラフの start ノードはスレッドの開始を表し，end ノードはスレッドの終了を表す．

図 1 は 2 つのスタックの操作を行うプログラムの例である．Stack クラスのオブジェクト s1，s2 を生成し，s1 を初期化し，外部データを s2 に読み込む．その後，s2 の全データを逆順に s1 へ格納し，最後に s1 のデータを順に取り出す．このプログラムから得られた実行履歴を表 1 と表 2 とする．図 2 と図 3 は，この実行履歴を用いて作成したオブジェクト s1 と s2 についての DOPG である．

表 1 の実行履歴からオブジェクト s1 についての DOPG が作成される過程を説明する．まず s1 は 4 行目でコンストラクタが呼ばれ生成されるので，create ノードができる．次に，6 行目で init メソッドが呼ばれるので，call ノードができる．次に s1 は s2 の reverse メソッド内の 17 行目で push メソッドが呼ばれ，call ノードができる．もう一度，push メソッドが 17 行目で呼ばれるが，17 行目での push メソッド呼出しを表す call ノードがすでに存在するので，その call ノードへのエッジがひかれる．s2 の reverse メソッドから復帰した後，do ブロック内の 10 行目で pop メソッドが呼ばれ call ノードができる．while 文の条件式で empty メソッドが呼ばれ call のノードができる．次に 10 行目で pop メソッドが呼ばれるが，これに対応するノードは既に存在するので，その call ノードへのエッジがひかれる．最後に 12 行目で empty メソッドが呼ばれるが，これに対応するノードに対応するノードが既に存在し，その call ノードへのエッジも既に存在するので，何も追加しない．

次にオブジェクト s2 についての DOPG が作成される過程について述べる．s2 はまず 5 行目で生成されるので，create ノードができる．次に 7 行目で read メソッドが呼ばれ call ノードができる．次に 8 行目で reverse メソッドが呼ばれ，call ノードができる．reverse メソッド内でのイベントは別グラフで表現される．reverse メソッドのグラフは entry ノードで始まる．呼出し元の call ノードからこの entry ノードへ call エッジがひかれる．次に 16 行目で empty メソッドが呼ばれ call ノードができ，entry ノードから seq エッジでつながれる．次に 17 行目で pop メソッドが呼ばれ call ノードができる．次に empty メソッドが呼ばれるが既に対応するノードが存在するのでその call ノードへエッジがひかれる．続く pop メソッド

```
1  class Stack{
2        public static void main(String[] argv){
3              int i = 0;
4              Stack s1 = new Stack();
5              Stack s2 = new Stack();
6              s1.init();
7              s2.read();
8              s2.reverse(s1);
9              do{
10                   s1.pop();
11                   i = i + 1;
12             }while(!s1.empty());
13       }
14
15       public void reverse(Stack to){
16             while(!empty()){
17                   to.push(pop());
18             }
19       }
20
21       ...
22 }
```

図 1: ソースコード例

と empty メソッドも同様にノード作成とエッジ作成が省略される．ここで reverse メソッドから復帰するので，return ノードができ，return ノードから呼出し元の call ノードで return ノードがひかれる．16 行目の empty メソッドを表す call ノードの次に return ノードができる．reverse メソッドから復帰した後の命令は reverse メソッドの call ノードの次に seq エッジでつながれるが，ここでオブジェクトへの命令呼出しは終わるので，end ノードがつながれる．

表 1: 図 1 のプログラムの実行から得られた s1 についての実行履歴例

実行順序	呼出し命令	位置
1	Stack	Stack.java#L.4
2	init	Stack.java#L.6
3	push	Stack.java#L.17
4	push	Stack.java#L.17
5	pop	Stack.java#L.10
6	empty	Stack.java#L.12
7	pop	Stack.java#L.10
8	empty	Stack.java#L.12

表 2: 図 1 のプログラムの実行から得られた s2 についての実行履歴例

実行順序	呼出し命令	位置
1	Stack	Stack.java#L.5
2	read	Stack.java#L.7
3	reverse	Stack.java#L.8
4	empty	Stack.java#L.16
5	pop	Stack.java#L.17
6	empty	Stack.java#L.16
7	pop	Stack.java#L.17
8	empty	Stack.java#L.16

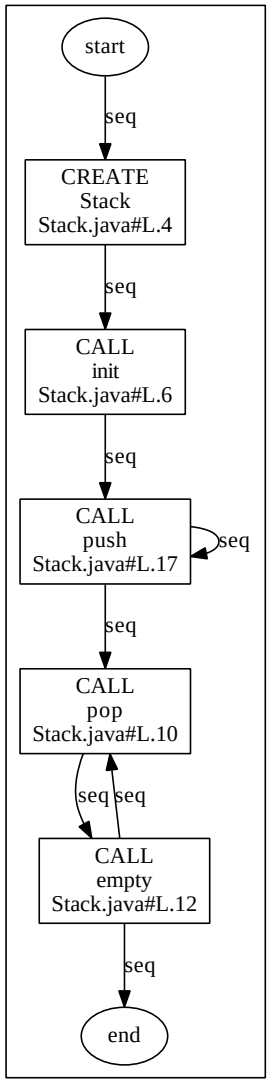

図 2: s1 についての DOPG

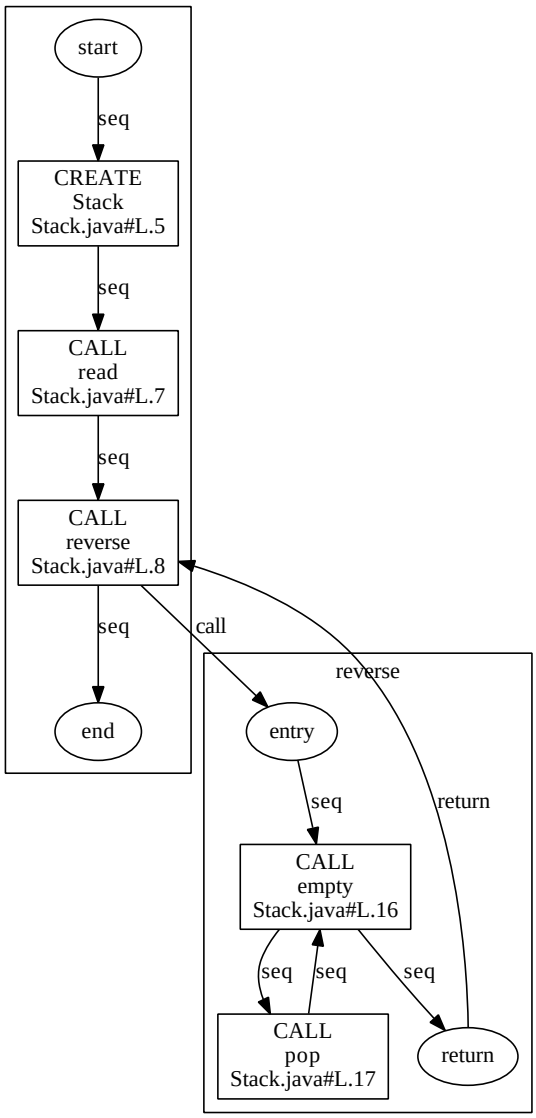

図 3: s2 についての DOPG

3　オブジェクトの振舞い予測手法

提案するオブジェクトの振舞いを予測する手法について説明する．提案手法では，予測対象のプログラムをあらかじめ実行し，オブジェクトの振舞い表現するオートマトンを事例として，取得する．その後，もう一度プログラムを実行する際に出現したオブジェクトの振舞いと事例との比較を行い，現オブジェクトと事例の振舞いを対応付ける．そして，対応付けられた事例を利用して振舞いを予測する．

本研究でいうオブジェクトの振舞いはオブジェクトの生成命令とそのオブジェクトをレシーバーオブジェクトとするメソッド呼出し命令の列である．プログラムの実行履歴を本手法の入力とし，利用者は予測されるオブジェクトのメソッド呼出し順序を得ることができる．

3.1　事例の取得

まず予測対象のプログラムを実行し，実行履歴を取得する，取得する実行履歴はオブジェクトの生成命令とオブジェクトに対するメソッド呼出し命令とそれらの位置の列である．また，各命令のソースコードでの位置も記録する．取得した実行履歴はオブジェクトごとに分け，DOPG の形にまとめる．形状の同じである DOPG が複数存在する場合には，それは 1 つの DOPG と見なす.

次に DOPG から，オブジェクトの振舞いを表現するオートマトンを作成する．オートマトンはオブジェクト生成命令やメソッド呼出し命令のイベントを入力として状態遷移する．DOPG の start ノードから end ノードまでの一連のノード列が表すイベントを入力として受取ると受理状態となる．オートマトン作成は次のプロセスで行う.

1. DOPG のメソッドごとのグラフを除去する

   DOPG に含まれるメソッドごとのグラフを除去する．除去は次の手順で行う．まず，複数の call ノードから call，return エッジでつながれるメソッドのグラフを call ノードの数だけ複製し，1 つの call ノードにつき 1 つのメソッドのグラフが存在するようにする．次に，メソッドのグラフの entry ノードと seq エッジでつながれるノードへ，メソッド呼出し元の call ノードから seq エッジをひく．また，メソッドのグラフの return ノードと seq エッジでつながれるノードから，メソッド呼出し元の call ノードへ seq エッジをひく．次に，entry ノードと return ノードを取り除き，それとつながっている seq エッジも取り除く．さらに，メソッド呼出し元の call ノードから出る，seq エッジを取り除く．ただしこのメソッドのグラフを除去する手順の中で追加した seq エッジは取り除かない．最後にメソッドのグラフを取り除く．図 4 の DOPG は，A メソッドでのイベントを表すグラフがあり，また A メソッドが異なる 2 か所で呼び出されてい

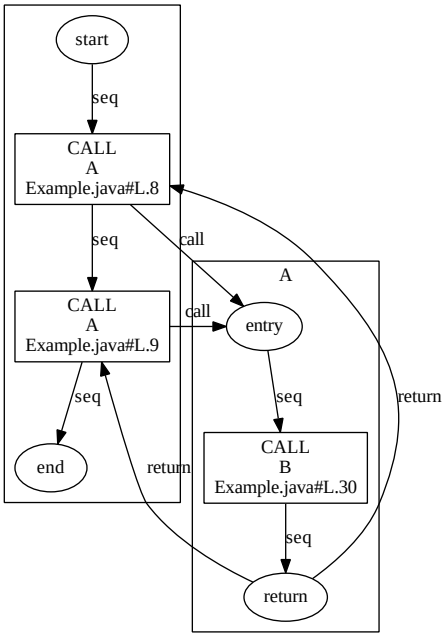

図 4: メソッドのグラフの除去前

る例である．まず，複数の call ノードからつながれている A メソッドのグラフを呼出しの数だけ複製し，メソッドのグラフが 1 つの呼出し元の call ノードにのみつながっている状態にする (図 5)．次に，メソッド A のグラフへの call エッジを持つ call ノードと，メソッド A の entry ノードと seq エッジでつながっているノードを seq エッジでつなぐ．また，メソッド A のグラフ中の return ノードへの seq エッジを持つノードと，呼出し元の call ノードと seq エッジでつながっているノードを seq エッジでつなぐ (図 6)．そして，entry ノードと return ノードを取り除き，つながっていたエッジも取り除く．また，呼出し元の call ノードから出る seq エッジを取り除く (図 7)．最後にメソッド A のグラフを除去する (図 8)．

2. オートマトンの状態を定義する

   end ノードを除く各ノードごとに対応する固有の状態をオートマトンに定義する．

3. オートマトンの状態遷移を定義する

   end ノードを除く DOPG のノードについて，その中のあるノードを A とする．A から出る seq エッジの行先のノードを B とする．B の命令を入力とする，A に対応する状態から B に対応する状態への状態遷移をオートマトンに定義する．end ノードを除く全てのノードに対してこの処理を行う．

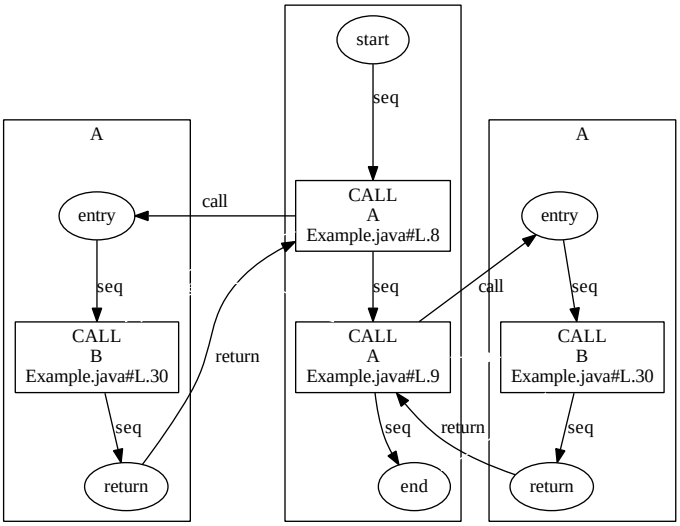

図 5: メソッドのグラフの複製

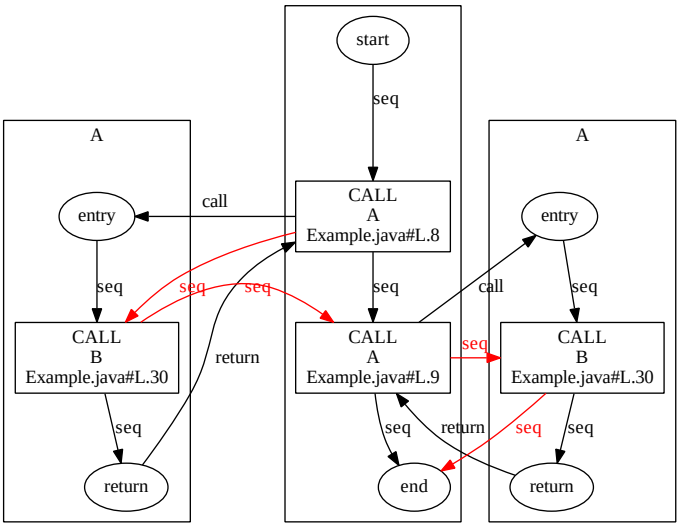

図 6: seq エッジを追加

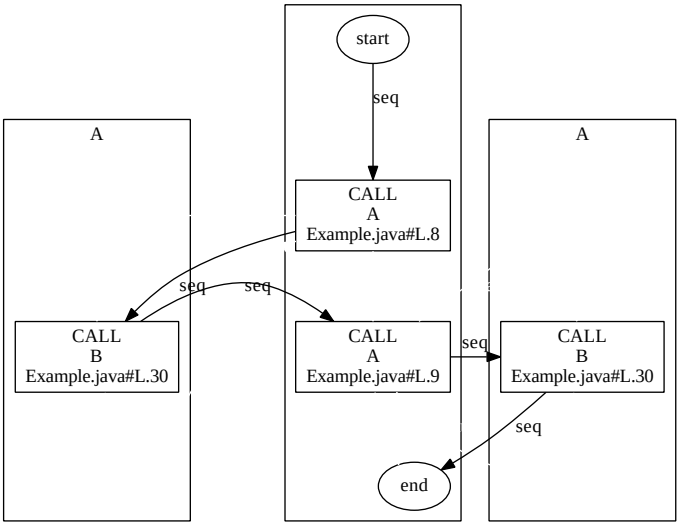

図 7: entry ノードと return ノードとそれとつながるエッジの除去及び呼出し元の call ノードから出る seq エッジの除去

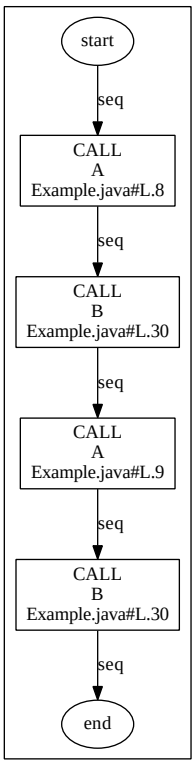

図 8: メソッドのグラフの除去後

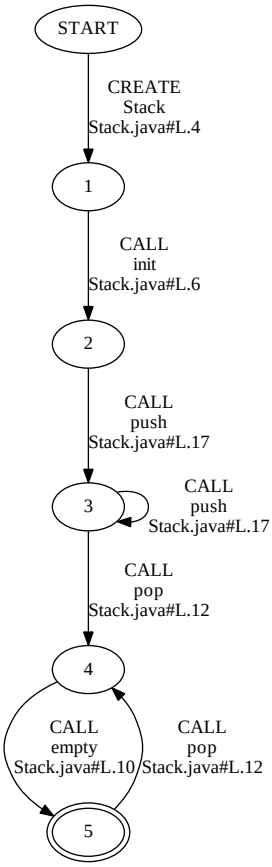

図 9: s1 に対応するオートマトン

4. オートマトンの初期状態・受理状態を決定する

   start ノードに対応する状態をオートマトンの初期状態とし，end ノードへのエッジを持つノードに対応する状態をオートマトンの受理状態とする．

以上のプロセスによって，図 2 と図 3 の DOPG をオートマトンに変換したものがそれぞれ図 9 と図 10 である．なお，オートマトンではノードが状態を表し，エッジが状態遷移を表している．また，START とラベルの付いたノードが初期状態で，二重線のノードが受理状態を表す.

この方法において，再帰が含まれる DOPG は，オートマトン作成プロセスにおいてサブルーチンの除去を行うことができず，対応できない．また，複数のスレッドで使用されるオブジェクトは，全てのスレッドでのオブジェクトの使用期間が重ならない場合にのみ対応する．スレッドごとの DOPG をそれぞれオートマトンに変換した後，先に実行されたスレッドの DOPG に対応するオートマトンの受理状態と，後のオートマトンの初期状態をマージ

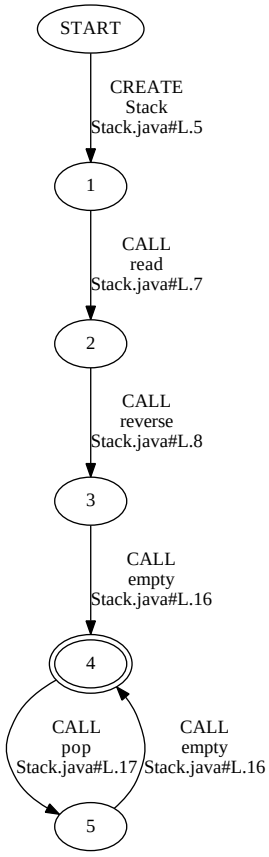

図 10: s2 に対応するオートマトン

し，2 つのオートマトンを 1 つのオートマトンにする．初期状態は先のオートマトンの初期状態となり，受理状態は後のオートマトンの受理状態となる．複数のスレッドから同時に使用されるオブジェクトについては対応しない．

3.2 振舞いの一致する事例の検索

プログラムをもう一度実行し，実行中に出現した各オブジェクトについて，振舞いの一致する事例を検索する．

まず，出現したあるオブジェクトについて，そのオブジェクトが属するクラスのオートマトン全てを振舞いの一致する事例の候補とする．出現したオブジェクトに対しての命令があったとき，候補のオートマトン全てに命令を入力として与え，状態遷移させる．ここで状態遷移できないオートマトンがあるならば，それを候補から除外する．

この状態遷移と，候補からの除外をオブジェクトに対する命令がある度に行う．最終的に，候補のオートマトンが 1 つになったとき，残ったオートマトンを振舞いの一致する事例であると決定する．

たとえば，図 1 を一度実行して事例を取得したあと，もう一度実行したときに Stack クラスのオブジェクト s1 の振舞いと一致する事例を検索する場合，最初は図 9 と図 10 のオートマトンが候補となる．ここではこれらのオートマトンをそれぞれ A_1，A_2 と呼ぶことにする．オブジェクト s1 についての実行履歴は図 1 のものと同一であるとする．まず候補の 2 つのオートマトンに，生成命令を入力として与える．この生成命令はソースコードの 4 行目で起こったもので，A_1 は状態遷移する．しかし，A_2 は受理状態へ到達し得なくなるので，これを候補から除外する．この時点で，候補が A_1 の 1 つになったので，これを振舞いの一致するオートマトンと決定する．

3.3 振舞いの予測方法

事例のオートマトンと対応付けされたオブジェクトについて，対応づけられた時点でのオートマトンの状態から受理状態までに受け取り得るメソッド呼出し列を，予測するメソッド呼出しの列とする．3.2 節で示した例において，予測されるオブジェクト s1 の振舞いは状態 1 から状態 5 までのメソッド呼出しとなる．

4 評価実験

提案する手法によりどの程度振舞いの予測が可能かを評価するために評価実験を行った．評価実験では，実際のアプリケーションから手法で用いるオートマトンを取得し，そのオートマトンを検索に必要なメソッド呼出し数と予測できる命令の数の観点から調査した．また，事例を取得する実行と，事例との対応付けを行う再度の実行で，プログラムが同じ動作をすると仮定した．

実験対象のプログラムは DaCapo ベンチマーク [5] とした．DaCapo ベンチマークは，多数の Java 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．実行時にはそれぞれのアプリケーションで実行の規模を small，default，large から指定できる．

4.1 評価方法

評価実験では，DaCapo ベンチマークを用いてアプリケーションを実行し，手法で用いるオートマトンを作成した．対象としたアプリケーションは avrora，batik，lusearch，pmd，xalan の 5 つで，実行の規模を small とした．ここでは，評価実験の手順について説明する．

4.1.1 実行履歴および DOPG の取得とオートマトンの作成

実行履歴の取得には，Amida [1] を使用した．Amida は実行履歴を取得する対象のクラスを指定できる．実験では，実験対象のそれぞれのアプリケーションで表 3 のクラスを対象とした．なお，表中の $*$ はワイルドカードを表す．

また，Amida で作成した DOPG をオートマトンに変換するツール作成し，それを使用してオートマトンの作成を行った．

表 3: DaCapo ベンチマークのアプリケーションごとの対象とするクラス

アプリケーション名	対象クラス
avrora	avrora.$*$
batik	org.apache.batik.$*$
lusearch	org.apache.lucene.$*$
pmd	net.sourceforge.pmd.$*$
xalan	org.apache.xalan.$*$

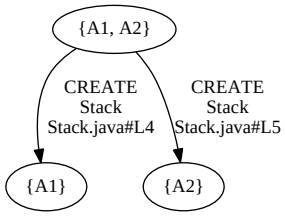

図 11: 生存オートマトン木の例

4.1.2 オートマトンの調査

オートマトンの性質を調査するために，クラス単位での生存オートマトン木を作成した．生存オートマトン木は，ノードが生存オートマトンの集合を表し，エッジがオブジェクト生成命令やメソッド呼出し命令を表すグラフである．この木の根は，クラスの全オートマトンを初期状態で持つ集合である．あるノードの子は，オートマトンへの入力の数だけ子を持ち，子は入力を受取った後のオートマトン集合を持つ．このとき，受理状態へ到達し得ないオートマトンを集合から取り除く．ノードが持つオートマトンの数が 1 つのとき，そのノードを葉ノードとする．また，ループ構造を表現するオートマトンの存在により，木の深さが無限になる．そこで，オートマトン集合の要素数が等しく全てのオートマトンの状態が等しい祖先が存在するノードについては，その子孫にあたるノードを無視する．

図 11 に生存オートマトン木の例を示す．例は図 1 の Stack クラスについての木である．s1，s2 の振舞いを表すオートマトンをそれぞれ A1，A2 としている．木の根は，2 種類のオートマトンを集合に持つ．そこから 2 種類の生成命令により分岐し，それぞれを入力として受取ったオートマトンを集合に含む子ができる．この子はともに生存オートマトン集合の要素数が 1 なので，葉となる．

この生存オートマトン木を用いて，以下の値を計測した．

- $Trace$:振舞いの検索に必要なメソッド呼出し数

  振舞いの一致するオートマトンを検索するために必要な生成命令とメソッド呼出し命令の数である．$Trace$ の値が小さいほど，実行中のプログラムのオブジェクトと，事例との対応付けが少ない命令呼出しで行える．生存オートマトン木における根から葉までの距離がこれに対応する．ただし，ここでの葉とは，生存オートマトン集合の要素数が 1 のものを指す．

- $Predict$:予測可能なメソッド呼出し数

  振舞いの一致するオートマトンの検索が終わったのちに，手法により予測できるメソッド呼出し列の長さである．検索ができた後の状態から受理状態に至るまでの最小のメ

ソッド呼出し数をこの数値として用いた.

これらの値を用いて，オートマトンのライフサイクルに占める予測部の割合 R を次のように定義した.

$$R = \frac{Predict}{Trace + Predict}$$

4.1.3 除外したクラス

提案手法と作成したツールで対応できない以下のクラスを実験の対象から除外した．全 1015 クラスのうち 212 クラスが該当した.

- 複数のスレッドから同時に使用されるオブジェクトを含むクラス
- 再帰を含むオブジェクトを含むクラス
- オートマトンの状態数が 50 を超えるオブジェクトを含むクラス

  状態数の大きいオートマトンを含むクラスの生存オートマトン木を作成することができなかったため，オートマトンの状態数に制限を設けた.

4.2 結果

実験ではクラスごとに $Trace$ と $Predict$ の平均値を求め，それらを用いて R の値を計測した．結果からは，90%のクラスは予測システムを実装した際に，予測が行えることがわかった．結果を図 12 に示す．このグラフは，求めた R の値でクラスを降順に並べたものである.

グラフから，約 66%のクラスで $R = 1$ となっていることがわかる．これらのクラスは DOPG によってオブジェクトの振舞いを分類したとき，ただ 1 種類の振舞いにまとめられる．オブジェクトの振舞いを分析するとき，複数のオブジェクトから特定の振舞いをするものを選ぶ必要がない．本手法により予測を行う場合には，事例との対応付けに 1 つの命令呼出しも必要としないので，これらのクラスはプログラムを実行しクラスのオブジェクトが出現した時点で振舞いが予測可能である.

約 24%のクラスでは R の値が $0 < R < 1$ となっている．これらのクラスでは，オブジェクトの振舞いが 2 種類以上存在し，本手法で振舞いの検索を行うには，1 つ以上の命令呼出しが必要となる．これらのクラスでは，少なくとも 1 つ以上のメソッド呼出し命令が予測できることが多いことがわかった．図 13，図 14 はそれぞれ，$0 < R < 1$ であるクラスでの全ての $Trace$ の値と $Predict$ の値を表す．これはクラスの生存オートマトン木の全ての葉に

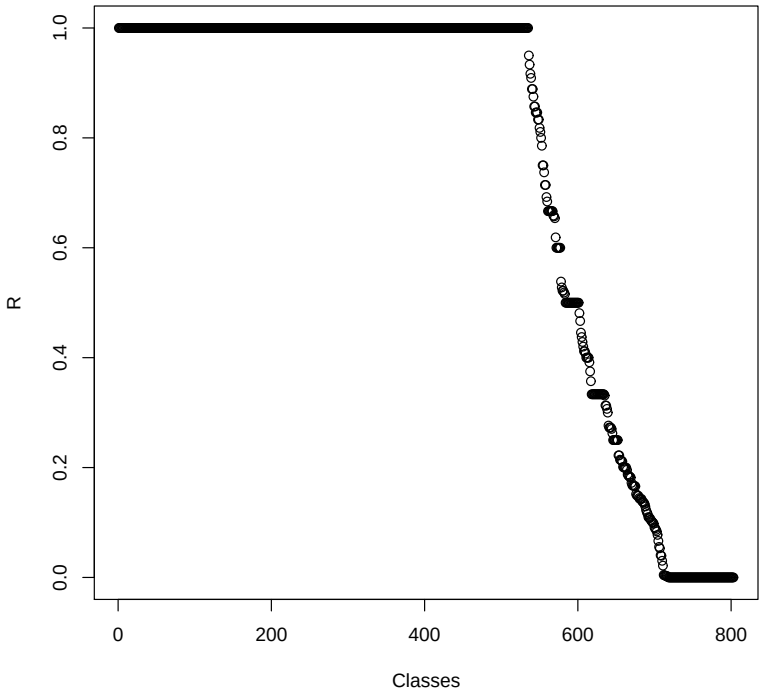

図 12: クラスごとに計測した R の値

対応する $Predict$ の値を含む．このグラフから，88%の $Predict$ の値が 1 以上となっていることがわかる．これらのクラスでの $Trace$ および $Predict$ の値の基本統計量を，それぞれ表 5 と表 4 に示す．$Predict$ の値の平均値から，これらのクラスでは約 4 つの命令呼出しが予測可能であることが期待できることがわかる．

約 10%のクラスでは $R = 0$ となっている．これらのクラスに属するオートマトンは全て，識別された時点で受理状態に達している．

また，全てのクラスでの $Predict$ の平均は約 4.08 であった．つまり，手法により約 4 つのメソッド呼出し命令の予測が期待できる．

以上の結果から，$R = 0$ となった約 90%のクラスでは，プログラム実行中にオブジェクトの振舞いを予測することが可能であることがわかる．つまり，デバッガに本研究の提案手法による予測システムを組み込むことで，プログラム実行中にオブジェクトの振舞いを表示することが可能であることがわかる．また，オブジェクトの動作を分析する際に，複数のオブジェクトから特定の振舞いをするオブジェクトを選択する必要がある $R < 1$ となるクラスは，全体の約 34%であり，予測可能なクラスは全体の約 24%である．つまり予測が必要なクラスの約%70 が予測である．したがって，この予測手法により，特定の振舞いをするオブジェクトを選択できることが期待できる．

次に $Trace$ の値や $Predct$ の値の特徴について述べる．図 15 は全ての生存オートマトン木の葉における $Trace$ の値と $Predict$ の値を表示したものである．この図から，$Trace$ の値に多数の種類がある一方，$Predict$ の値が偏っている部分があることがわかる．たとえば，$Predict$ の値が 12，14，17，21 である部分に現れている．これらは同じクラスの生存オートマトン木にある葉に対応する値であり，あるオートマトンの識別に必要なメソッド呼出し列には多数の種類があるが，識別後の状態はいくつかに偏ることを示している．

表 4: $0 < R < 1$ であるクラスの $Predict$ の値の基本統計量

最小値	中央値	平均値	最大値
0.0	2.0	4.1	34.0

表 5: $0 < R < 1$ であるクラスの $Trace$ の値の基本統計量

最小値	中央値	平均値	最大値
1.0	32.0	38.4	190.0

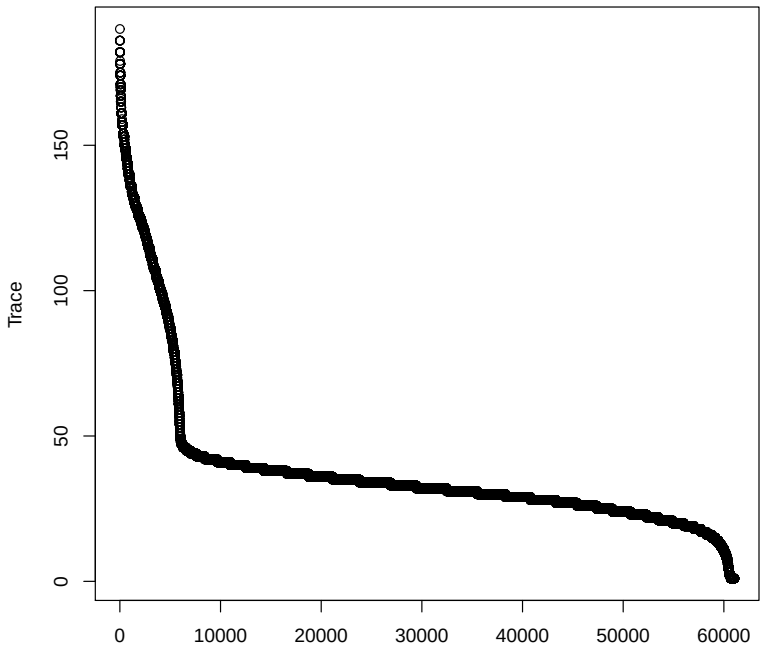

図 13: $0 < R < 1$ であるクラスの全ての $Trace$ の値

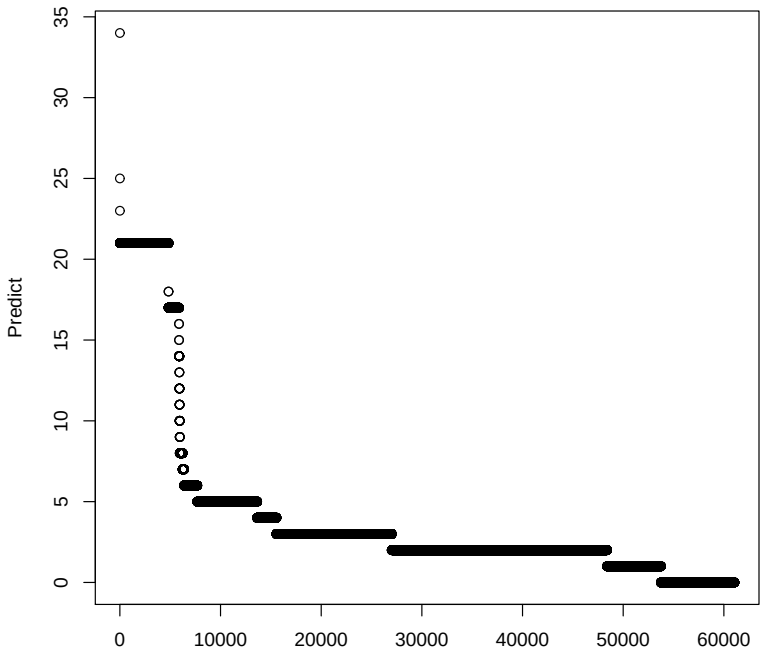

図 14: $0 < R < 1$ であるクラスの全ての $Predict$ の値

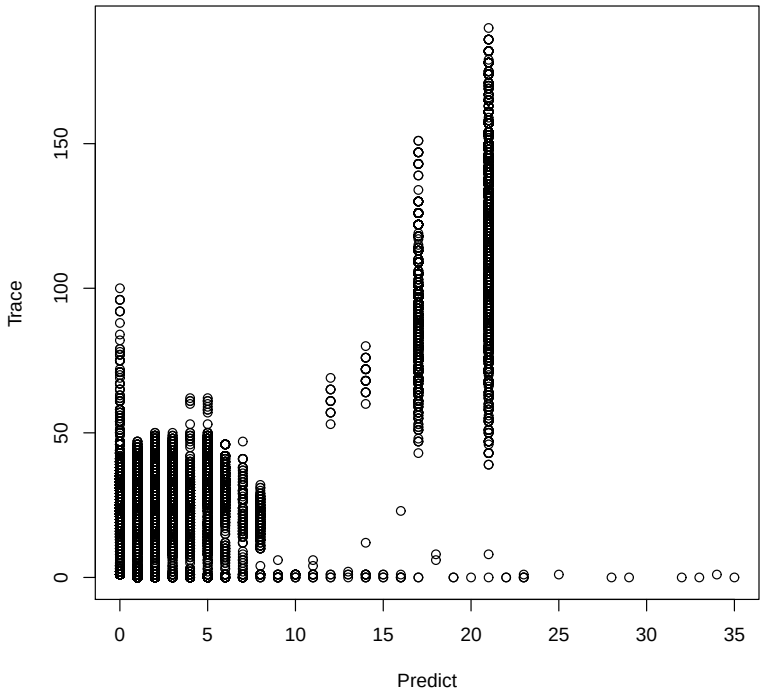

図 15: $Trace$ と $Predict$

5　関連研究

プログラムの状態を予測する試みも既存研究として，Michail ら [8]，Buse ら [4]，Mytkowicz ら [10] の研究がある．Michail らは，同一のバグを回避するため，GUI プログラムでイベントや関数呼出しの履歴とバグ報告をあらかじめ蓄積することで既知のバグに遭遇することを予測する手法を提案している．Buse らは機械学習を用いて，ソースコードからパスの通過確率を予測する手法を提案している．Mytkowicz らは現在実行中の関数とスタックポインタの位置から，現在のコールスタックの内容を特定する．提案手法では，オブジェクトが今後取り得る振舞いを推測しており，より粒度の小さい推測を可能としている．

動的解析を用いて，情報の絞り込みを行っている研究として，Nainar ら [2]，Zhang ら [14] のものがある．Nainar らは，デバッグのオーバーヘッドを減らすため，動的解析により監視する部分を最初は小さく設定し，その後，デバッグに必要な範囲まで広げていくことによって絞り込んでいる．Zhang らは，一度目に実行した際の実行履歴に対し，目的とする処理に関係のない部分を削除することで目的とする処理を理解するために必要な実行履歴を絞り込んでいる．本稿で提案している手法では，一度実行した際に抽出できた振舞いに基づき，対象となるオブジェクトを絞り込んでいる．これは，既存の研究に比べ粒度が小さいため，プログラムの詳細な理解に適していると考える．

Lo ら [6] は複数の実行履歴から振舞いモデルを生成する手法を提案している．Lo らの手法では，複数の実行履歴からそれら全てに当てはまるルールを抽出し，それらのルールを統合することにより振る舞いモデルを表すオートマトンを作成する．本稿で提案している手法では，各オブジェクトの実行履歴ごとに DOPG を経由してオートマトンを作成している．そのため，一度しか起こっていない振る舞いに対してもオートマトンを作成でき，発生頻度の少ない振舞いでも推測できる．

6　まとめ

本研究では，あらかじめ取得した事例との比較を行うことによるオブジェクトの振舞いを予測する手法を提案した．本手法は，一度プログラムを実行し実行履歴を取得し，それを用いてオブジェクトごとの DOPG を作成し，それをオートマトンに変換し，オブジェクトの振舞いの事例とする．そして，もう一度プログラムを実行したときに出現したオブジェクトの振舞いと，振舞いが一致する事例のオートマトンを検索し，そのオートマトンを利用してオブジェクトの振舞いの予測を行う．

また，本手法によりどの程度オブジェクトの振舞いの予測が可能かを調査するために評価実験を行った．評価実験では，DaCapo ベンチマークの 803 個のクラスのうち約 34%のクラスで，分析したいオブジェクトの選択を行う必要があり，その約 70%のクラスでオブジェクトの振舞いを予測可能であることがわかった．また，平均で 4 つの命令呼出しが予測可能であることがわかった．この結果から，デバッガに本手法による予測システムを組み込むことで，予測を行うことが可能で，それがプログラムにおけるオブジェクトの振舞いの分析に利用できることが期待できる．

今後の課題としては，状態の多いオートマトンへの対処や，同時に複数のスレッドから使用されるオブジェクトへの対処が挙げられる．また，手法の評価実験の範囲を広げ，様々な種類のアプリケーションで予測ができることを確認することが必要である．さらに，この手法によってオブジェクトの先の振舞いを知ることがオブジェクトの振舞いの分析に有効であるかどうかを調査することも必要である．

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　眞鍋 雄貴 特任助教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] Amida. http://sel.ist.osaka-u.ac.jp/˜ishio/amida/.

[2] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 255–264, 2010.

[3] Lionel. C. Briand, Yvan Labiche, and Jahanne. Leduc. Towards the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, Vol. 32, No. 9, pp. 642–663, 2006.

[4] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 144–154, 2009.

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

[6] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pp. 345–354, 2009.

[7] 前田直人. コンテキストを考慮したポインタ解析結果を用いたメソッド呼出しパターン検査. コンピュータソフトウェア, Vol. 26, No. 2, pp. 157–169, 2009.

[8] Amir Michail and Tao Xie. Helping users avoid bugs in gui applications. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pp. 107–116, 2005.

[9] 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.

[10] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. *In Proceedings of the 17th IEEE International Conference on Object-Oriented Programing, Systems, and Applications, OOPSLA'09*, pp. 175–190, October 2009.

[11] Jochen Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Proceedings of the 16th IEEE International Conference on Program Comprehension ICPC'08*, pp. 73–82, 2008.

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

[13] Atanes. Rountev, Olga. Volgin, and Miriam. Reddoch. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE'05*, pp. 96–102, 2005.

[14] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pp. 81–91, 2006.