DOPG を用いたオブジェクトの振舞い予測手法

Object Behavior Prediction Using Dynamic Object Process Graph

脇阪 大輝* 眞鍋 雄貴† 石尾 隆‡ 井上 克郎§

あらまし デバッガは開発者が欠陥の原因を調べるために重要なツールの 1 つである．デバッガのステップ実行によって，開発者はプログラムの振舞いを詳細に分析することができる．しかし，プログラムのある特定の振舞いを分析するためには，開発者はその振舞いの開始を予測し，振舞いが起きる直前に，プログラムの実行を一時停止しなくてはならない．本研究では，この作業を支援するために，オブジェクト指向プログラムを対象として，記録した実行履歴と現在の実行とを比較することでオブジェクトの振舞いを予測する手法を提案する．本手法は，過去の実行履歴から Dynamic Object Process Graph（DOPG）を抽出し，オブジェクトを操作するメソッド呼び出し文の系列を表現する決定性有限オートマトン（DFA）を構築する．プログラムのそれ以降の実行において，観測されたオブジェクトの振舞いに合致する DFA を特定することで，それぞれのオブジェクトの振舞いを予測する．DaCapo ベンチマークを用いた評価実験では，多くのクラスについて，呼び出し文の系列が予測可能であることを示した．

Summary. Debugger is an important tool for developers to investigate the cause of a failure. Single step execution of a debugger enables developers to analyze the behavior of objects in detail. However, to investigate a particular behavior using single step execution, developers have to predict the behavior and suspend its execution. In this paper, we propose to predict the behavior of objects by comparing the current execution with a set of recorded execution traces. Our approach extracts a set of Dynamic Object Process Graphs (DOPGs) from execution traces and translates them into deterministic finite automata (DFAs) representing the behavior of objects. During another execution, we predict the behavior of each object by identifying a DFA which matches the observed behavior of the object. To evaluate whether our approach can predict the behavior of objects or not, we have analyzed classes in the DaCapo benchmarks as a preliminary experiment. The result shows that a sequence of method call sites for each instance is predictable for many classes.

1 はじめに

オブジェクト指向プログラムは，多数のオブジェクトを使用してその機能を実行する．プログラムの欠陥を分析するとき，開発者は欠陥に関与している可能性のあるオブジェクトを特定し，その振舞いを詳細に調査しなければならない．そのような作業を支援するため，多数のデバッガでブレイクポイントやステップ実行などの機能が提供されている．これらの機能を用いると，開発者はプログラムの実行経路やメモリの状態を調べることができ，欠陥の原因を効果的に調査することが可能である．一方で，デバッガの機能を活用する上での困難の 1 つは，プログラムの実行を適切なタイミングで停止することである．Omniscient Debugger [1] のようにプログラムのすべての状態を保存するアプローチを取らない限り，プログラムの実行時の状態を任意の時点に巻き戻すことはできない．そのため，開発者は，分析した

*Hiroki Wakisaka，大阪大学大学院情報科学研究科

†Yuki Manabe，大阪大学大学院情報科学研究科

‡Takashi Ishio，大阪大学大学院情報科学研究科

§Katsuro Inoue，大阪大学大学院情報科学研究科

```
while(…){
        // 繰り返しごとに異なるオブジェクト
        Object obj = getInstance(…);

        /* 調査したいプログラム文 */
        …
        /*                          */

        if(…){
                // このメソッド呼出しを実行するオブジェクトに興味がある
                obj.interestObject();
        }
}
```

図 1　複数のインスタンスが関与するプログラム文の例

い振舞いが実行された後から，詳細な分析作業を開始することはできない．そこで，開発者は，プログラムの振舞いを予測し，分析したい振舞いが実行されるよりも前に，ブレイクポイントを配置する必要がある．

ブレイクポイントはプログラム文を指定して配置されるもので，一般に，オブジェクトを区別することができない．オブジェクト指向プログラムでは，たとえ同じクラスのインスタンスであっても，関連付けられるデータやユーザの操作によって，特定のオブジェクトだけが他のオブジェクトとは異なる機能を実行することがある [2]．そのため，開発者が分析したいプログラム文にブレークポイントを配置しただけでは，開発者にとって興味のある振舞いを行うインスタンスだけでなく，同じプログラム文を実行する他のすべてのインスタンスに対してもプログラムが一時停止してしまうことになる．プログラムの詳細な振舞いをステップ実行によって確認することを目的としている場合，無関係なインスタンスについても個別にステップ実行を行うことになり，開発者にとっては大きな負担となる． 例えば，図 1 のようなプログラムの実行を詳細に調査することを考える．変数 obj から参照されるオブジェクトが繰り返しによって異なり，開発者は interestObject メソッドを実行するオブジェクトが関与する実行の調査を行いたいとする．このとき，開発者は調査したいプログラム文の先頭でプログラムを停止するが，この時点では obj によって参照されるオブジェクトが興味のあるオブジェクトであるかどうかはわからない．よって開発者は無関係なオブジェクトについてもステップ実行を行うことになる．interestObject メソッドが呼び出される文でプログラムを停止することで興味のあるオブジェクトのみを抽出できるが，この時点で調査したいプログラム文を既に通過しているため調査を行うことができない．

本研究では，興味のある振舞いを行うインスタンスかどうかをあらかじめ判断できるように， 記録した実行履歴と現在の実行を比較することによって，オブジェクトの振舞いを予測する手法を提案する．プログラムのあるテストケースを二度実行したとき，一度目の実行におけるオブジェクトの振舞いが，二度目の実行においても観測されると仮定し，一度目の実行と二度目の実行のオブジェクトの動作を対応付ける．具体的な手法としては，実行履歴に含まれた各オブジェクトの振舞いを，メソッド呼出し文を入力として状態遷移を行う決定性有限オートマトン（DFA）として抽出する．現在の実行で観測された各オブジェクトの振舞いに一致する DFA を特定することで，それらのオブジェクトが，次にどのメソッド呼出し文から，呼び出しを受け取る可能性があるかを特定する．これにより，開発者は分析したい振舞いを表現した DFA に合致しているオブジェクトが出現したときのみ，プログラムの実行の詳細を分析することが可能になる． 図 1 の例において，開発者は，プロ

グラムの調査を開始したい地点でプログラムを停止し，その地点における obj から参照されるオブジェクトが interestObject メソッドを呼び出すオブジェクトであるかを知ることができる．これにより，無関係なオブジェクトについてステップ実行することを避けることが可能である．

オブジェクトの振舞いを開発者にとって有用なレベルで予測するため，本研究では，オブジェクトの振舞いモデルとして Dynamic Object Process Graph（DOPG）[3] を作成し，それを DFA に変換する．DFA によるモデル化は既存手法でも行われおり，それに倣っている． DOPG は，ある 1 つのオブジェクトの振舞いを表現する有向グラフであり，各ノードはオブジェクトが受け取ったメソッド呼出しのソースコード位置を，各エッジはノード間の制御フローを，それぞれ表現する．2 つのオブジェクトの DOPG は，2 つのオブジェクトが同じメソッド呼出し文の系列によって操作された場合に一致する．単純な呼び出し文の系列と異なる点は，DOPG はループ文によるメソッド呼出しの回数の違いを取り除いてオブジェクトの振舞いを比較することである． 本手法では，実行履歴から DOPG を抽出し，DOPG のノードを DFA の状態遷移に変換することで，将来受け取ることが期待されるメソッド呼び出し文の系列を予測する．

本手法がオブジェクトの振舞いを予測できるかどうかを評価するため，評価実験を行った．予測能力の指標としては，新しい実行において，オブジェクトが対応する DFA を特定するまでに必要なメソッド呼出し文の個数，特定された後にそのオブジェクトが受け取ると予測できるメソッド呼出し文の個数を用いた．実験対象は，DaCapo ベンチマーク[1]に含まれる 5 つのアプリケーションの 1015 個のクラスである．マルチスレッドによるアクセスなど，提案手法では取り扱えないオブジェクトを含むクラスを除いた 803 クラスに対して本手法を適用した結果，535 クラスについては，各クラスについて，1 つの DOPG によってオブジェクトの振舞いが表現された．これらのクラスの二度目の実行におけるメソッド呼出しは，記録したメソッド呼出しと同じであると予測できる．また，183 個のクラスにおいては，平均で 4 つのメソッド呼出しが予測でき，残る 85 個のクラスについては，すべてのメソッド呼び出し系列が得られない限り，あるオブジェクトが所属する DFA を特定することができなかった．

デバッガに組み込んだ場合の利便性については別途評価が必要であるが，本論文の貢献は，DOPG に基づく DFA を構築することでオブジェクトのメソッド呼び出し文を予測する手法を示したことと，多くのクラスにおいて予測が可能であることを実験で示したことである．

2 関連研究

Ressia ら [4] は，デバッグにおいて，オブジェクト単位での情報分析が重要であることを指摘し，クラスの各インスタンスを明確に区別して，振舞いの系列や状態を確認できる Object-Centric Debugging を提案している．オブジェクト単位の振舞いに着目するという点は共通しているが，オブジェクトの振舞いを予測するというのが，我々の研究の新規性となっている．

プログラムの振舞いを予測するという観点での関連研究としては，Michail ら [5] の手法が挙げられる．この手法は，GUI イベントとメソッド呼出しを結び付け，情報検索技術を使用して，ユーザの操作によってアプリケーションの障害が発生する可能性を予測する．この手法では，ユーザを保護するという観点から，アプリケーションレベルの機能単位での振舞いの予測に注力しており，誤った予測に基づく障害の警告を発する可能性がある．デバッグ作業において誤った予測は開発者に悪影響を与える可能性があるため，我々の研究では，決定性のアルゴリズムを用いて，オブジェクトに関連する既知の振舞いを特定できたときにのみ，未来のメソッド呼

[1] The DaCapo Benchmark Suite. http://dacapobench.org/

出しを出力する形式を採用している．

Lewis [1] は，システムの振舞いを開発者が分析できるようにするため，Omniscient Debugger を提案した．この手法では，プログラムの振舞いを再生し分析できるように，すべての実行履歴を記録する．そのため，実行履歴の記録に関するコストは高いが，単体で動作するシステムの分析には効果的である．一方で，実行履歴に含まれないような状態，たとえば外部システムの状態は再現できず，外部システムとの相互作用の分析などには適用しにくい．本研究では，分析したい相互作用を実行履歴から再現するかわりに，その相互作用が開始すると思われる場面でプログラムの実行を停止することで，詳細な分析を行う機会を開発者に提供することを目指している．また，Omniscient Debugger と連携することで，分析対象だと思われる振舞いに対してのみ詳細な実行履歴の記録を開始することも可能になると考えている．

プログラムの実行から，オブジェクトの振舞いモデルを推定し，実行を監視する手法は，多くの既存研究でも行われている．たとえば，Lo ら [6] は複数の実行履歴からクラスのインスタンスの共通する振舞いを抽出する手法を提案した．Lee ら [7] は，オブジェクトの集合に対して振舞いモデルを抽出することを提案した．これらの手法は，オートマトンとして表現されたモデル内部で等価だと思われる状態をマージし，クラスごとにただ 1 つのモデルを抽出する．本研究の目標は振舞いモデルを出力することではないため，DFA の集合のマージは行わず，オブジェクトごとに個別のモデルを抽出する．

Bodden ら [8] は，静的解析によって，オブジェクトが特定の振舞いを行うソースコードの位置を特定し，オブジェクトの振舞いを監視するコストを低減する手法を提案している．この種の静的解析は，本研究においても実行時性能を向上するために非常に重要であるが，現在の実装では動的解析のみを使用している．

3 振舞い予測手法

本研究では，記録されたオブジェクト単位の振舞いを利用して，現在の実行において観測されるオブジェクトのこれからの振舞いを予測する手法を提案する．提案手法の手順は，次の通りである．

1. 対象のプログラムをテストケースを用いて実行し，実行履歴を記録する．
2. 実行履歴から，オブジェクトの振舞いのモデルとして DFA を抽出する．このステップはオフライン解析である．
3. 対象のプログラムを再度実行し，オンライン解析を行う．各オブジェクトについて，振舞いを観測することで，各実行時点で，オブジェクトに対するイベント系列を受理しうる DFA を特定する．そのような DFA が存在すれば，これから起きるオブジェクトに対してのイベント列をその DFA から予測することができる．予測された振舞いは，デバッガがプログラムの実行を停止した際に開発者に提示する情報であり，また，条件付きブレイクポイントの条件式などにも利用できる情報となる．

これ以降，提案手法の各ステップを説明するために，図 2 に示すソースコード例を用いる．この図において，ソースコードの行番号は，コメントにて示している．具体的には，7 行目が Example オブジェクトの生成，8 行目と 9 行目がメソッド A の呼出し，また，30 行目がメソッド A からメソッド B への呼出しとなっている．

3.1 実行履歴の取得

本研究における実行履歴とは，Java プログラムの実行時イベントの系列であり，3 種類のイベント *call*，*entry*，*return* からなる．*call* イベントはメソッドとコンストラクタの呼出しを表す．*entry* イベントはメソッドの実行の開始を，*return* イベントはメソッドの実行の終了を，それぞれ表す．イベント系列における i 番目の

```
class Example {
        ...
        public static void main(String[] argv){
                Example ex = new Example(); /* L.7 */
                ex.A(); /* L.8 */
                ex.A(); /* L.9 */
        }
        ...
        public void A(){
                ...
                B(); /* L.30 */
        }
        ...
        public void B(){...}
}
```

図 2　ソースコード例

表 1　図 2 のプログラムの実行履歴

k_i	M_i	l_i
call	`Example`	L.7
entry	`Example`	
exit	`Example`	
call	`A`	L.8
entry	`A`	
call	`B`	L.30
entry	`B`	
exit	`B`	
exit	`A`	
call	`A`	L.9
entry	`A`	
call	`B`	L.30
entry	`B`	
exit	`B`	
exit	`A`	

イベント e_i を，5 つの属性の組 $\langle k_i, t_i, o_i, M_i, l_i \rangle$ によって表現する．ここで，k_i は $\{call, entry, return\}$ のうちのいずれかに対応し，t_i はイベントが発生したスレッドの ID を表す．o_i はメソッドを呼び出されたオブジェクトの ID を，M_i は呼び出されたメソッドを表し，l_i はメソッド呼出し文のソースコードでの位置を表す．プログラムを複数回実行して実行履歴を収集した場合，スレッド ID とオブジェクト ID は，それぞれの実行において，異なる ID を割り当てられると仮定する．

本研究では，オブジェクトごとの振舞いを表す DFA を抽出するため，まず，実行履歴から，特定のオブジェクトに関するイベント系列のみを抽出する．この時，オブジェクト o に対する実行時イベント列を $T(o) = \{e_i | o_i = o\}$ とする．

図 2 のソースコードを実行した場合，1 つの Example オブジェクトを 1 つのスレッド上で生成する．そのオブジェクトに対する実行履歴の情報を表 1 に示す．この表では，イベントの種別（k_i），呼び出されたメソッド（M_i）とソースコード位置 (l_i) のみを示しており，オブジェクト ID とスレッド ID は省略した．表の最初の行が，コードの 7 行目でのコンストラクタ呼出しに対応し，次の *entry* イベントがそのコンストラクタの実行の開始に対応する，というように実行履歴は記録されている．

3.2　DFA の抽出

DOPG は，オブジェクトがプログラム内で初めて使用されてから，最後に使用されるまでの生存期間の振舞いを表す有向グラフである．DOPG は 5 種類のノード，*call*，*entry*，*return*，*start*，*end* を持つ．*start* ノードと *end* ノードはそれぞれオブジェクトの使用開始と終了を表す．他の 3 つのノードは，オブジェクトに関係するイベントが発生したソースコード位置に対応する．DOPG のこれらのノードは，3 種類の辺，*call*，*return*，*seq* により接続される．*call* 辺と *return* 辺はメソッド呼出しを表しており，*call* 辺は *call* ノードから *entry* ノードに，*return* 辺は *return* ノードから *call* ノードに，それぞれ接続される．DOPG は実行履歴から抽出されるため，*call* ノードは，その呼び出しに対応して実行されたメソッドへの *call* 辺を持つことになる．一方，*seq* 辺はノード間の実行順序を表しており，*call* ノードと *entry* ノードは少なくとも 1 つの *seq* 辺を持つ．また，DOPG の各ノードはソースコード位置で識別されるため，ループにより同じ位置でのメソッド呼出しが繰り返し実行された場合には，自身への *seq* エッジを持つ *call* ノードとして表現される．

オブジェクト o に対する DOPG は実行履歴 $T(o)$ から，Quante の定義 [3] に基づいて抽出する．図 3 のグラフは表 1 の実行履歴から抽出した DOPG の例である．こ

の DOPG は，メソッド A の呼出し位置が異なるため，2 つのメソッド A に対する *call* ノードを持つ．メソッド B は 2 回呼び出されるが，同一の呼び出し文によるものであるため，1 つの *call* ノードによって表現される．

本手法では，DOPG を以下のステップにより DFA に変換する．

1. DOPG 内のメソッド呼出しをインライン展開する．
   DOPG では，メソッドを表す部分グラフが複数の *call* ノードから共有されるので，そのような部分グラフをインライン展開により除去する．これは，文脈自由文法により表現される入れ子のメソッド呼出しを，DFA が扱えないためである．具体的な手順であるが，まず，それぞれの *call* ノードに対して，部分グラフを複製する．図 3 の DOPG からは，図 4 のグラフが得られる．次に，手続き間の制御フローを表す *seq* エッジを追加する．*call* ノード n_c から *entry* ノード n_e への *call* 辺に対して，n_c から，n_e の次に実行されるノード（*seq* 辺で接続されているノード）への直接の *seq* 辺を追加する．*return* ノード n_r から *call* ノード n_c への各 *return* 辺に対しても，n_r の直前のノードから，n_c の次に実行されるノードへの *seq* 辺を追加する．図 5 に，このステップの結果を示す．最後に，*entry* ノード，*return* ノード，*call* 辺，*return* 辺を削除する．これにより，図 6 に示すような，メソッドのインライン展開が完了した DOPG が得られる．
2. DFA の状態を作成する．
   end ノードを除く，DOPG の各ノードに，それぞれ状態を対応づける．
3. DFA の状態遷移を作成する．
   DOPG に含まれるすべての *seq* 辺について，辺が接続するノードにそれぞれ対応する状態間で，状態遷移を作成する．遷移するための入力は，辺の到達先ノードが表現するメソッド呼出し文である．DOPG の *end* ノードへとつながる *seq* 辺については無視する．
4. sink 状態を追加する．
   DFA が受理し得ないイベント系列が到着した場合を対応する sink 状態を追加する．作成した状態遷移に関係しない入力は，すべて sink 状態への遷移として扱う．
5. 初期状態と受理状態を決定する．
   DFA の初期状態は，DOPG の *start* ノードに対応する状態とする．受理状態は，DOPG の *end* ノードへの辺を持つノードに対応する状態とする．図 7 のグラフは，図 6 のインライン展開済み DOPG から得られる DFA である．ただし，図中では，sink 状態は省略している．

DFA の抽出は，すべてのオブジェクトに対して行う．あるクラス C のすべてのインスタンスに対して得られた DFA の中から，互いに完全一致した DFA を除去して得られるクラス C の DFA 集合を $\mathbb{A}_C = \{A_1, A_2, ..., A_n\}$ と記述する．

3.3 振舞いの予測

本手法では，抽出した DFA を用いて，対象プログラムの実行中におけるオブジェクトの振舞いを予測する．このオンライン分析では，各オブジェクトに対して，DFA 集合 $\mathbb{A}_C$ のコピーを作成する．そしてオブジェクトごとのメソッド呼出し位置を観測し，すべての DFA に入力として与える．現在観測しているオブジェクトの振舞いに対応する DFA A_k は，以下の条件式 (1) を判定することで得られる．ただし，$s(A, l)$ は入力 l を与えられたときの DFA A の状態を表す．

$$s(A_k, l) \neq sink \quad \wedge \quad (\forall i. i \neq k \rightarrow s(A_i, l) = sink) \tag{1}$$

この条件は，ただ 1 つの DFA A_k だけが，観測されたイベントに続くイベントを受理し得るときに満たされる．本手法の出力は，各オブジェクトについての，A_k とその現在の状態 $s(A_k, l)$ となる．デバッガは，この情報を用いて，オブジェクトの

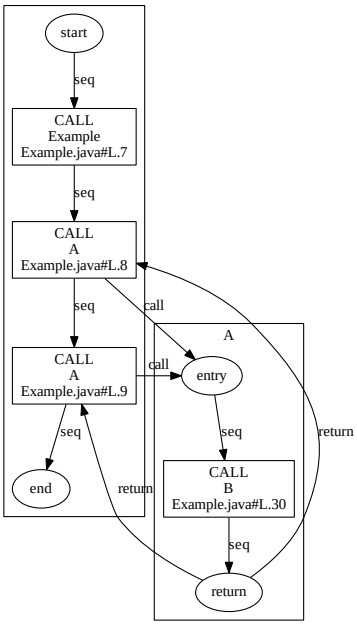

図 3　表 1 の実行履歴に対応する DOPG

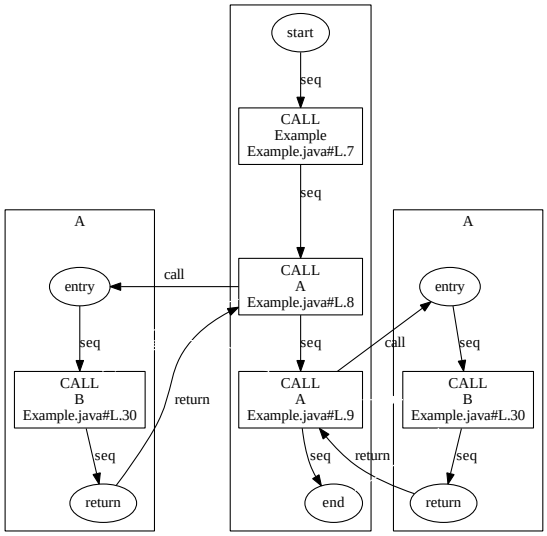

図 4　メソッド呼出しごとの部分グラフを複製した DOPG

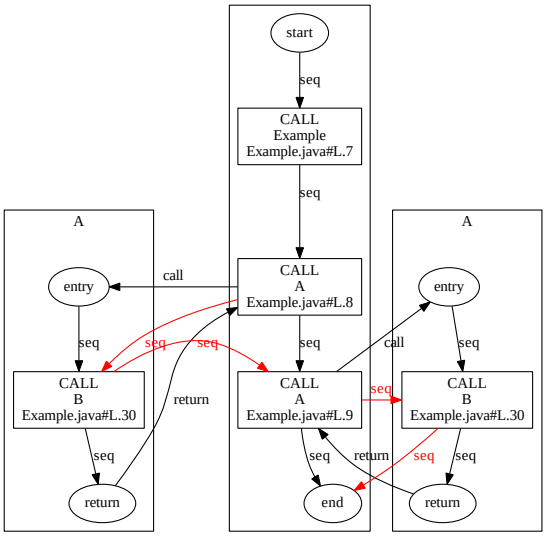

図 5　*call*，*return* 辺に対応する *seq* 辺を追加した状態

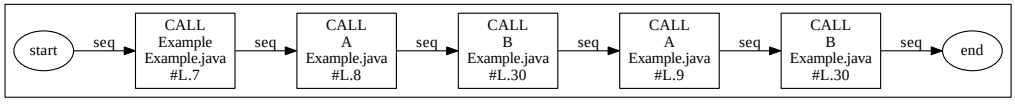

図 6　インライン展開が完了した DOPG

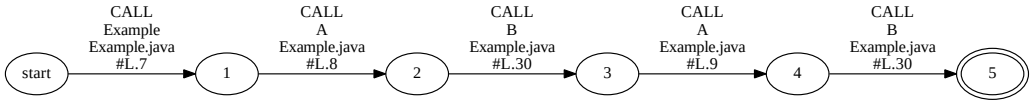

図 7　図 6 の DOPG に対応する DFA

状態や，今後の振舞いを開発者に提示することになる．条件を満たす（これからのイベント系列を受理しうる）DFA が複数残っている場合には，まだ DFA の特定が完了していないため，そのオブジェクトについての予測は出力しない．また，これからのイベントを受理し得る DFA がなくなった場合，そのイベント列を未知の振舞いであると判定する．

3.4　制限

任意のマルチスレッド実行は扱えない．本手法は，マルチスレッドプログラムで使用されているオブジェクトの振舞いの予測については，特定の条件を満たすものにのみ，経験的な方法で対応している．その条件とは，オブジェクト o に対してスレッド t_i，t_j が逐次的にアクセスするというものであり，具体的には，以下に示す条件式 (2) を満たすことである．

$$begin(t_i, o) > end(t_j, o) \quad \vee \quad end(t_i, o) < begin(t_j, o) \tag{2}$$

ここで $begin(t, o)$ はスレッド t におけるオブジェクト o に関する最初のイベントの時刻を表し，$end(t, o)$ は最後のイベントの時刻を表す．この仮定により，あるスレッドで初期化され，その後別のスレッドでアクセスされるオブジェクトを許容する．つまり，マルチスレッドでの逐次アクセスを，シングルスレッドでのイベント列とみなしている．あるクラスにおいて，1 つでも逐次的にアクセスされていないインスタンスがある場合，そのクラスを解析対象から除外する．

再帰呼び出しは扱えない．本手法ではメソッドの再帰呼出しを扱うことができない．DFA では，文脈自由文法で表現される再帰呼び出しを含むイベント系列を扱うことができないためである．本手法では，あるクラスのインスタンスが 1 つでも再帰呼出しを受ける場合，そのクラスを解析対象から除外する．

未知の振舞いは予測の誤りにつながる．本手法では，ある時点よりも未来に発生するイベント系列を受理し得る DFA を，観測されたイベントに基づいて特定する．しかし，これからのオブジェクトの振舞いを保証するわけではない．未知の振舞いが観測された場合，特定された DFA によってそのイベント列が受理されない可能性がある．この場合，手法の出力は，誤った予測であったことになる．

イベントの予測系列は一意に定まるわけではない．DFA は，メソッド呼出しの繰り返しを状態遷移のループによって表現するため，その繰り返しの回数などは予測することができない．

4　評価実験

本手法によりオブジェクトの振舞いが予測できるかどうかを調査するため，Javassist [2]を用いてプロトタイプを作成し，評価実験を行った．実験対象は，DaCapo ベンチマークに含まれる 5 つのアプリケーション，avrora，batik，lusearch，pmd，xalan の実行履歴である．アプリケーションには 1015 のクラスが含まれており，そのうち以下の条件に当てはまるオブジェクトが含まれる 212 クラスを対象から除外した．

- 複数のスレッドから同時にアクセスされるオブジェクト
- メソッドの再帰呼出しを含むオブジェクト

[2] Javassist. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

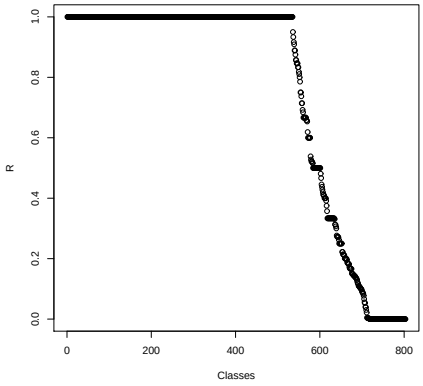

図 8　クラスと $R(C)$ の値の関係．クラスは $R(C)$ の値の降順に並べている．

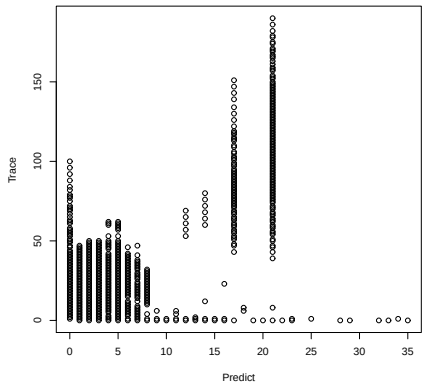

図 9　DFA の特定に必要なイベント数と，予測できるイベント数の関係

- DFA の状態が 50 を超えるオブジェクト

1 つ目と 2 つ目の条件は本手法の制限を反映している．3 つ目の条件は大域的に使用されるオブジェクト（ユーティリティ [9]）を除外するためのものである．

実験対象の各クラスに対して，予測できるイベント数の期待値を評価するため，メトリクス $Trace(C)$，$Predict(C)$，$R(C)$ を定義した．

- $Trace(C)$ は DFA を特定するために必要なメソッド呼出し数の平均を表す．$\mathbb{A}_C = \{A_1, A_2, ..., A_n\}$ の各 DFA について，A_k を特定するために必要なイベント数（複数のオートマトンに共通する状態遷移数+1）を l_k としたとき，l_k の平均値が $Trace(C)$ となる．
- $Predict(C)$ は予測可能なメソッド呼出し文の数を表す．$\mathbb{A}$ の各 DFA について，l_k 個のイベントによって DFA が特定された後，A_k の受理状態に到達するまでの最短のイベント列の長さを p_k としたとき，$Predict(C)$ は p_k の平均値である．
- $R(C)$ はオブジェクトの生存区間全体における予測可能な振舞いの比率であり，$R(C) = Predict(C)/(Trace(C) + Predict(C))$ で求める．

各クラスに対して，オフライン解析で得られたオートマトン集合を解析し，$Trace(C)$，$Predict(C)$，$R(C)$ を計算した．図 8 は，横軸にクラスを $R(C)$ の値で降順に並べている．図 9 は各 DFA の l_k と p_k の値を示したもので，縦軸は DFA を特定するために必要なメソッド呼出しの数を，横軸はその DFA の受理状態までの最短経路の長さを表している．

実験結果から，535 個のクラス（解析したクラスの 66%）で $R(C) = 1$ となった．これらのクラスは 1 つの DOPG で振舞いが表現される，すなわち，特定のメソッド呼出し順によって使用されるクラスである．Java では，アプリケーションの機能の実行を 1 つのオブジェクトが担当することが多いため，このカテゴリに多くのクラスが含まれている．たとえば，batik の `DisplayManager` クラスはシステムのディスプレイのプロパティを管理しており，また，avrora の `ClockDomain` クラスはシステム全体のデータセットを管理している．これらのクラスに対しては，本手法により，DFA 全体を，予測可能な振舞いとして示すことができる．

次に，183 クラス（24%）で $R(C) > 0$ となった．これらのクラスは 2 つ以上の DFA を持ち，平均で 4 つのメソッド呼出しを予測することができる．これらのクラスで共通する性質はわかっていないが，`set` メソッド等により異なる設定を外部から受け取り，その後は同じ使い方をされるというクラスがいくつか発見された．た

とえば，batik の `StyleSheet` クラスや `PathParser` クラスが，このカテゴリに含まれている．これらのクラスでは，オブジェクトごとに様々な l_k の値を待つが，p_k の値は同一となっている．このことが，図 9 において垂直な線として確認できる．本手法では，これらのクラスについては，初期化処理が完了した後の共通する振舞いを予測することができる．

残る 85 個のクラス（10%）では $R(C) = 0$ となった．これらのクラスでは，オブジェクトが最後のイベントを受け取った時点で初めて，DFA を特定することができる．たとえば，`org.apache.batik.dom` パッケージの `GenericText` クラスは，初期化等の処理はオブジェクト間で同一であるが，使われ方だけが異なることから，このカテゴリに含まれていた．これらのクラスに対しては，本手法は効果を持たないが，既存の手法 [6] により 1 つの DFA にマージすることで，途中までに共通するメソッド呼出し系列を得られると考えられる．$R(C)$ の値はオートマトンの形状から計算できるため，クラスごとに異なる手法を適用することは容易である．

5 まとめ

本研究では，予め記録した実行履歴から既知の振舞いを特定することにより，プログラムの実行中に多くのオブジェクトの振舞いを予測できることを示した．今後の課題として，振舞いの予測が，実際のデバッグ作業にどのような影響を与えるかを評価することが挙げられる．まず，振舞いの予測によるデバッグ支援が効果的である場面が実際にどの程度あるか，DOPG が 1 つのクラスについての予測がデバッグ作業に有益であるかの調査を行いたいと考えている．また，再帰呼出しやマルチスレッドを使用したプログラムに対応できるように，手法を拡張することが挙げられる．再帰呼出しは DFA をプッシュダウンオートマトンで表現することにより，マルチスレッドはスレッドごとのオートマトンの直積によって状態を表現することにより，それぞれ対応できると考えている．

謝辞 本研究は，科研費・若手研究 (A)（課題番号:23680001）および科研費・若手研究（B）（課題番号:24700029） の助成を得た．

参考文献

[1] B. Lewis. Debugging backwards in time. In *Proceedings of the 5th International Workshop on Automated Debugging*, 2003.

[2] 宗像聡, 石尾隆, 井上克郎. 類似する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，第 163 回ソフトウェア工学研究発表会，第 31 巻, pp. 225–232, March 2009.

[3] J. Quante and R. Koschke. Dynamic object process graphs. *Journal of Systems and Software*, Vol. 81, pp. 481–501, 2008.

[4] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 485–495, 2012.

[5] A. Michail and T. Xie. Helping users avoid bugs in GUI applications. In *Proceedings of the 27th International Conference on Software Engineering*, pp. 107–116, 2005.

[6] D. Lo, L. Mariani, and M. 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*, pp. 345–354, 2009.

[7] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceeding of the 33rd International Conference on Software Engineering*, pp. 591–600, 2011.

[8] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 36–47, 2008.

[9] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 181–190, 2006.