社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2007-55(2007-12)

オブジェクト指向プログラムの実行履歴に対する機能単位での自動分割手法

渡邊　結†　石尾　隆†　井上　克郎†

† 大阪大学大学院情報科学研究科
E-mail: †{wtnb-y,ishio,inoue}@ist.osaka-u.ac.jp

あらまし　オブジェクト指向プログラムによるシステムの振る舞いは，動的束縛等によって実行時に決定される要素が多いため，その理解にはプログラムの実行履歴のような動的情報を解析し，シーケンス図として可視化することが有効である．しかし，実行履歴中には複数の機能の実行が含まれるため，シーケンス図のサイズは履歴の長さに比例して巨大化し，そのままでは人間の読解に不適切である．そこで本研究では，オブジェクト指向言語プログラムの実行履歴を，キャッシュアルゴリズムとメソッド呼び出し情報を用いてフェイズ単位で分割する手法を提案する．
キーワード　Javaプログラム，実行履歴，動的解析

Detecting functional division in execution trace of object oriented program

Yui WATANABE†, Takashi ISHIO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
E-mail: †{wtnb-y,ishio,inoue}@ist.osaka-u.ac.jp

Abstract　An object-oriented program involves dynamically determined elements such as dynamic binding and exception handling. To understand such a system, it is effective to analyze execution traces and show them as sequence diagrams. However, a raw trace including several phases is too large to be tranceformed into a single sequence diagram. In this paper, we propose a method to detect functional division in an execution trace of a program by using a cache algorithm and method call infomation.
Key words　Java program, execution trace, dynamic analysis

1. はじめに

オブジェクト指向プログラムでは，実行時に動的に生成されるオブジェクトが相互にメッセージを交換する事によってシステムが動作するが，どのオブジェクトがどのようにメッセージ通信を行うかは，実行時に動的に決定される．このようなシステムの振る舞いを理解するためには，プログラムの実行履歴を可視化することが有効である [19].

我々はこれまでに，実行プログラムのメソッド呼び出し履歴を取得し，ループや再帰呼び出し処理を検出，除去することで簡略化したシーケンス図を生成する手法を提案している [16].

しかし，取得した実行履歴には複数の機能の実行が重複込みで含まれるため，シーケンス図の各位置で，プログラムのどの機能の振る舞いが可視化されているかは開発者が読解する必要がある [10].

そこで本研究では，実行履歴を特定の機能を実行した時間軸上の区間（フェイズ）に自動的に分割する手法を提案する．提案手法は，オブジェクト指向プログラムの各機能が多数の作業用オブジェクト群によって実行されるという性質 [7] に基づき，動作するオブジェクトの集合（ワーキングセット）の変化をLRUキャッシュアルゴリズムを用いて検出する．開発者が認識している機能単位が，ワーキングセットの変化によって検出できることを，実験によって確認した．

本稿では，提案手法のアルゴリズムと，適用実験の結果を述べる．2. 節ではプログラムの振る舞いの可視化とフェイズの識別について述べ，3. 節で実行履歴をフェイズ単位で自動的に分割する手法について，4. 節で提案手法を実際に既存のソフトウェアシステムに適用した実験結果について説明する．最後に，5. 節で適用実験の考察，6. 節でまとめを行う．

2. 背　　景

2.1 プログラムの振る舞いの可視化

オブジェクト指向プログラミングでは，クラスを用いて処理の抽象化を行い，また継承や多態性などを利用して，オブジェクト間のメソッド呼び出しとしてソフトウェアの機能を実現する．その一方で，開発者がソフトウェアの機能の詳細を読解する場合，たとえばあるメソッド呼び出し文が，動的束縛の結果，実際に呼び出しうる複数のメソッド定義を発見するといった労力の増大が生じており，支援ツールの必要性が指摘されている [6], [15], [19].

特定の機能を実現するためのオブジェクトの相互作用は，設計段階において，主に UML のシーケンス図によって記述される [8]．シーケンス図は，縦軸に時間を，横軸にオブジェクトを取り，各オブジェクトが時系列に従ってメソッド呼び出し（メッセージ通信）を行う動作シナリオを記述する．このような動作シナリオは，そのソフトウェアにおける特定のタスクを実現する方法を記述しており，ソフトウェアの理解や再利用にも適した単位である [12].

しかし，設計図を活用することが困難な場合もある．たとえば，開発が進行していく中でソフトウェアの機能に変更が加えられたとき，開発者が設計図の更新を怠ると，設計図がソフトウェアの最新の状態を正しく反映しなくなる [5]．また，ソフトウェアを実装していく段階で，設計図には登場しないようなクラス，メソッドが追加されることもある [16].

そこで，開発されたソフトウェアからオブジェクト間の相互作用を検出し，UML のシーケンス図を含む様々な形式によってとして可視化する方法が研究されている．ソースコードを解析する手法では，クラス間での呼び出し関係を可視化する手法 [4] や，オブジェクトの生成文からのデータフローを解析し，どこで生成されたオブジェクトが動作するかを可視化する手法 [17], 制御フローグラフからシーケンス図を生成する手法 [13]，登場するオブジェクトに適切な名前を変数名から抽出する手法 [14] などが提案されている．

ソースコードを用いた解析によって得られる結果は，動的束縛などを静的に解決できる範囲に限られる．そのため，実際にソフトウェアを実行し，どのオブジェクトが呼び出されたかを記録した実行履歴 *(Execution Trace)* を解析し，可視化する手法が広く研究されている．

実行履歴は，プログラムの実行開始から終了までの，膨大な数のメソッド呼び出しの系列である．オブジェクトの ID およびメソッド名の系列があれば，シーケンス図として可視化することができる [16]．しかし，単純にシーケンス図として可視化しただけでは人間の読解に適したサイズにはならず，パターン検出を用いた圧縮処理 [11] や，ループや再帰呼び出しの検出 [16], 実装の詳細である可能性が高いメソッドの自動的なフィルタリング [3]，概要を把握するための縮小表示 [10] など，様々な手法が適用されている．

2.2 フェイズの識別

実行履歴を用いたシーケンス図の生成手法の多くは，実行履歴に含まれる処理の概要を把握することを主眼として提案されており，開発者が特定の処理の詳細を読解する用途には不向きである．しかし，1 つの実行履歴には，特定の処理を実行する時間軸上の区間であるフェイズ *(phase)* が複数存在していることが知られている [10]．たとえば，ソフトウェアのテスト実行時の履歴には，テスト用の事前処理および事後処理（テスト用データの生成や破棄など）が含まれる．これらの処理をシーケンス図から取り除くことが，テスト実行時のソフトウェアの振る舞いを分析する際に有用であったと報告されている [1]．このようなフェイズの認識は，特定の機能の詳細を読解する作業を補助するためのシーケンス図の生成にも有効であると期待される．

そこで本研究では，与えられた実行履歴からこのようなフェイズの認識を自動化し，1 つの実行履歴を，複数のフェイズの列へと自動分割する手法を提案する．既存研究であるテスト実行の可視化では，JUnit フレームワークを用いた実行であることが仮定されており，メソッド名やスタックの深さなど実装上の知識が使用されている [1] が，本研究では任意の実行履歴を対象としたフェイズの認識を行う．

具体的な方法としては，動作しているオブジェクト集合の変化に着目する [9]．実行履歴中の各時刻において，「最近使用されたオブジェクト集合」を Least Recently Used (LRU) キャッシュに記録しておき，ある一定時間（ウィンドウサイズ）で多数の新しいオブジェクトが出現した，新しいフェイズに変化したことを認識する．フェイズの変化が認識された時刻から，時間軸を遡って探索し，メソッド呼び出しスタックの深さに基づいて，そのフェイズの開始時刻を特定する．

このようなオブジェクトの入れ替わりは，オブジェクト指向プログラムにおいて，1 つの機能を実現する際に中間データ用のオブジェクトを生成することが多く，また 1 つの手続きの実行が終了した時点で多数のオブジェクトを破棄するという特徴に由来している [7]．大多数のオブジェクトの生存期間は短く，また，ある一定期間を生き残ったオブジェクトのほとんどはその後も破棄されないという性質は，世代別ガベージコレクションにも利用されている [18].

本研究で識別することを目指すフェイズには，2 つの粒度がある．1 つはソフトウェアに対する要求を実現するための機能 (Feature) [2] であり，もう 1 つは，機能の詳細な実行シナリオの各ステップである．たとえば「ユーザがシステムにログインする」,「検索結果を画面に表示する」など，開発者が保守作業時に認識する，意味のあるまとまりを目標とする．

3. 提 案 手 法

提案手法は，実行時のメソッド呼び出し系列と粒度制御パラメータを入力とし，実行履歴のに含まれる各フェイズの実行履歴上での開始時刻（以降，本稿では開始点と呼ぶ）の集合を出力する．

3.1 アルゴリズム

提案手法のアルゴリズムを図 1 に示す．

入力は，実行時のメソッド呼び出しイベント系列 E であり，

```
procedure  DiviideLog
                input : E = [e_1, e_2, ... e_last]
                input : csize, w, threshold, m
                output : P : set of eventID
(1)   C ← φ, P ← φ
(2)   for t in [1...last]
(3)     updated[t] = update(C, e_t.caller, e_t.callee)
(4)     if ratio(t) ≥ threshold
(5)       P ← IdentifyPhaseTransition(t)
(6)     end if
(7)   end for

function  updated(C, callerID, calleeID) : integer
1)    b1 ← C.update(callerID)
2)    b2 ← C.update(calleeID)
3)    if b1 ∨ b2
4)              return 1
5)    else
6)              return 0
7)    end if

function  IdentifyPhaseTransition(t) : integer
1)    x = max(t − m + 1, 1)
2)    min = x
3)    while x ≤ t
4)      if e_min.callstack ≥ e_x.callstack
5)        min = x
6)      end if
7)      inc(x)
8)    end while
9)    return min
```

図 1　実行履歴の分割アルゴリズム

その要素 e_t はイベント時刻 t におけるイベントオブジェクトである．この時，$e_t.caller$ はメソッド呼び出し元オブジェクトの ID を，$e_t.callee$ はメソッド呼び出し先オブジェクトの ID を，$e_t.caller.callstack$ はメソッド呼び出し時のコールスタックの深さを表す．また，出力となる集合 P は，重複を含まない配列であり，実行履歴上の時刻の列である．

手法は「フェイズ移行の検出」と「フェイズ開始点の決定」の二段階に分かれている．まず，「協調動作するオブジェクト群に着目した実行履歴分割手法 [9]」を用いて，実行履歴上で実行機能フェイズが移行したことを読み取る．次に，そこからフェイズの開始点を，各時刻のメソッド呼び出しスタックの深さに基づいて，履歴上で一点に決定する．そして最終的に出力される開始点系列により，実行履歴はフェイズ単位で分割される．

以下，各段階の具体的なアルゴリズムについて述べる．

3.1.1　フェイズ移行の検出

図 1 において，$ratio(t)$ の値を計算する (4) までの部分が，サイズ $csize$ のキャッシュC を用いたフェイズ移行検出アルゴリズムである．

この手法は，実行履歴上の各時刻 t でメソッド呼び出しに関与した（呼び出し元・呼び出し先の）オブジェクトの ID をキャッシュC で記憶していき，キャッシュC 中のオブジェクトの入れ替わり頻度を更新頻度 $ratio(t)$ として計測する（$last$ は実行履歴のサイズ）．

キャッシュの更新は *Least Recently Used* アルゴリズムに基づく．つまり，キャッシュC 内の要素にヒットした場合は，その要素の参照時刻を更新する．キャッシュC 内の要素ににヒットせずかつ空きもなければ，新しいオブジェクト ID を新たに追加し，代わりに最も参照時刻が古い要素を破棄する．

図 1(3) では，時刻 t におけるキャッシュC の更新有無 $updated[t]$ が，関数 $updated()$ で計算される．まず，キャッシュC へ時刻 t で動作したオブジェクトの ID ($e_t.Caller, e_t.Callee(t)$) を新しく追加する．このとき，キャッシュ$C$ に新たなオブジェクト ID が少なくとも 1 つ追加されたとき，キャッシュが更新されたとみなして 1 を返す．

そして図 1(4) では，$updated[t]$ をもとに時刻 t におけるキャッシュの更新頻度 $ratio(t)$ を次式によって計算する．

$$ratio(t) = \frac{\Sigma_{x=\max(1,t-w+1)}^{t} updated[x]}{w}$$

ここで，w は過去何回分のメソッド呼び出しにおける更新回数を平均するかを表すパラメータであり，以後ウィンドウサイズと呼ぶ．

動作するオブジェクト群が安定している時は，更新頻度 $ratio(t)$ の値が 0 に近くなり，動作するオブジェクト群が大きく入れ替わる時，更新頻度 $ratio(t)$ の値が 1 へ近づく．すなわち，更新頻度 $ratio(t)$ の値が高い時，フェイズの移行が生じていると判断できる．

3.1.2　フェイズ開始点の決定

フェイズが移行したことを示す「更新頻度 $ratio(t)$ の高い時刻」を，そのまま開始点として利用することはできない．これは，あるフェイズの移行に対応する更新頻度の高い時刻が，実行履歴上の一点でなく，複数のイベントにまたがって断続的に生じるためである．また，「更新頻度の高い時刻」は，動作するオブジェクト群が大きく入れ替わった際に表れるため，実際のフェイズ移行点からは必ず遅れて検出されるという特徴がある．

このため，「更新頻度の高い区間」を検出した後に，対応する機能フェイズの切り替わり点を，正しく，一点で決定しなければならない．ここで求めるフェイズ開始点は，フェイズが移行し始める最初の点，即ち，新しいフェイズで動作するオブジェクトの最初の 1 つが動作した時刻である．

ある機能フェイズが終了する時，その機能フェイズにおいて動作していたオブジェクトが順次動作を終える．更にその後，別の機能フェイズが開始するが，これは次の機能フェイズで動作するオブジェクトに対するメソッド呼び出しで開始される．従って，コールスタックの要素数（メソッド呼び出しの深さ）を実行履歴の時系列に沿って見たとき，それはフェイズの切り

替わる点において極小値であると考えられる．

そこで，更新頻度 $ratio(t)$ が閾値 $threshold$ 以上となる各時刻 t について，その時刻から過去 m 回のイベント中で，コールスタックの要素数が最小となる時刻を，出力するフェイズの開始点として決定する．ただし，区間内で該当する点が複数存在する場合は，時系列順で最新となる点を採用するものとする．

これにより，ある機能フェイズの移行に対応する「更新頻度の高い時刻」が実行履歴上で複数イベントにまたがって生じたとしても，それらから算出される開始点は一点に定まる．具体的なアルゴリズムは図1の関数 $IdentifyPhaseTransition$ に示す通りである．

ここで，m は過去何回分のメソッド呼び出しについてコールスタック要素数の最小値を検索するかの範囲を表す数であり，以後スコープサイズと呼ぶ．

3.2 分割を制御するパラメータ

提案手法には，入力として分割する実行履歴の他に，出力結果を制御する定数パラメータを渡す．パラメータは「キャッシュサイズ」「ウィンドウサイズ」「閾値」「スコープサイズ」の4つである．これらの値を変化させることで，出力される開始点の位置や個数が変化する．

a） キャッシュサイズ $csize$

「最近動作したオブジェクト群」を記録するキャッシュC の大きさを指定するパラメータであり，値は整数で最小で 1，最大で実行履歴上に登場するオブジェクトの数である．

1つの機能フェイズは，より小さい複数の機能のフェイズ実行の組み合わせからなるため，大まかに機能を分割する場合は，1つのフェイズで動作するオブジェクト集合の要素数も大きくなる．したがって，キャッシュサイズ値が大きいほど，より多くのオブジェクトを1つの集合としてみなすこととなり，実行履歴をより大まかな機能単位のフェイズで分割することになると考えられる．

b） ウィンドウサイズ w

「過去何回分のキャッシュ更新を平均するか」を指定するパラメータであり，値は整数で最小で 1，最大で実行履歴のイベント数 $last$ である．

このパラメータが大きいほど，直前のフェイズとの間で入れ替わるオブジェクト数が少ないフェイズの開始点は検出されにくくなる．すなわち，この値が大きいほど「より差異のある機能フェイズ間の切り替わり」だけを検出するようになり，逆に小さければ小規模で差異の小さいフェイズ移行も検出できるようになると考えられる．

c） 閾値 $threshold$

$ratio(t)$ の値からフェイズの移行を検知する閾値であり，値は最小で 0，最大で 1 である．

フェイズとフェイズの間で入れ替わるオブジェクトの数が多いほど，$ratio(t)$ の値は大きくなる．このため，閾値の値の変化はウィンドウサイズの値の変化と同様，大きいほど「より差異のある機能フェイズ間の切り替わり」だけを検出するようになり，逆に小さいほど，規模で差異の小さいフェイズ移行も検出できるようになると考えられる．

しかし，実際の $ratio(t)$ の値の最大値は，実行履歴の内容に加え，ウィンドウサイズに依存して上下するため，求めるフェイズ移行点付近の $ratio(t)$ だけを切り分けるような閾値を設定する事は難しい．

そこで 4. 節の適用実験では，閾値は出力分割点を制御するためではなく，誤検出を除去するために固定値で設定した．

d） スコープサイズ m

過去何回分のメソッド呼び出しについてコールスタック要素数の最小値を検索するかの範囲を表す数であり，値の範囲はウィンドウサイズと同様に最小で 1，最大で $last$ である．

実行履歴上で一つの機能（Feature）に対応する大きなフェイズが移行する点と，その機能を構成する小さなフェイズ間の移行点とでは，前者の移行点のコールスタック要素数の方がより小さいと予想される．このため，スコープサイズの値を大きくすると，実行履歴上でのイベント数が小さいフェイズの移行点に対応して $ratio(t)$ が上昇した場合であっても，そのフェイズの構成する Feature のフェイズ開始点を検出したり，フェイズの移行点ではないがたまたまコールスタック要素数が極小になっているようなイベントを誤検出する可能性が高まる．

4. 節の適用実験では，スコープサイズを解析対象である実行履歴の長さに比例した値として設定した．

4. 適用実験

本手法が認識するフェイズ境界が，開発者の判断と一致することを検証するため，適用実験を行った．

4.1 実験目的

本手法によって分割される各フェイズが，実際に開発者が判断する実行履歴上の機能を反映し，その開始点をを正しく一点で反映したものであることを確認する．

また，分割を制御するために実行履歴と共に与えられる2つのパラメータ「キャッシュサイズ」「ウィンドウサイズ」の値が，出力に与える影響を調査する．

4.2 実験方法

4.2.1 実験対象

実験対象として，業務用 web アプリケーション，ソースコード解析プログラム，書籍管理システムの3種の Java プログラムから取得した，複数の実行履歴を用いた．

4.2.2 実験手順

まず，対象システムより実行履歴を取得した実行履歴を人間が解析し，実行履歴上での正しいフェイズ開始時刻を事前に決定しておく．このとき指定されるフェイズの粒度は，ソフトウェアが実現する機能（Feature）レベルのフェイズと，それらの機能の実行をより詳細に記述した細粒度のフェイズとの二段階に分けられる．これは，各システムの実際の利用者・開発者に指定してもらった．

その上で，実行履歴に対して本手法を適用し，出力された開始点と，開発者が指定したフェイズ開始時刻との比較を行った．また，パラメータを変化させながら本手法を適用し，分割結果がどのように変化するかを調査した．

4.2.3 評価方法

a) パラメータによる分割粒度への影響

キャッシュサイズは実行履歴に登場する全オブジェクト数に対する割合として 5% 刻みで 5% から 95% まで，ウィンドウサイズは 5 から 200 まで，変化させながら本手法を適用し，出力される 20×20 通りのフェイズ開始点列を収集した．

なお，ウィンドウサイズを値域の最大まで変化させなかったのは，これ以上の値を用いても出力が変化しないためである．またこのとき，残る 2 つのパラメータ「閾値」「スコープサイズ」については，「閾値」を 0.10 に，「スコープサイズ」を履歴長 $last$ の 0.02 倍で固定して計算を行った．

b) 適合率と再現率

評価基準には，適合率と再現率を用いた．

提案手法が出力した開始点の集合を P，人間が指定したフェイズ開始時刻の集合を $Manual$ とし，また集合 S の要素数を $|S|$ と表記するとき，再現率と適合率は次式で定義される．

$$precision_{[\%]} = \frac{|P \cap Manual|}{|P|}, \quad recall_{[\%]} = \frac{|P \cap Manual|}{|Manual|}$$

4.3 実験結果

ここでは，業務用 web アプリケーションから取得した 1 つの実行履歴に対する実験結果を示す．このシステムは実際に運用されている，ソフトウェア開発ツールの貸出管理を行う web アプリケーションであり，規模はコードにして約 37000LOC である．実行履歴として取得した約 32000 イベントは 5 つの機能を順に実行したものであり，各機能を詳細に見た場合は 18 のステップ（重複を含む）が順に実行されている．

4.3.1 パラメータの影響

与えた 2 つのパラメータの値に対して，本手法が出力するフェイズ開始点の個数，すなわち実行履歴がいくつのフェイズに分割されるかを表す分割数の変化を図 2 に示す．

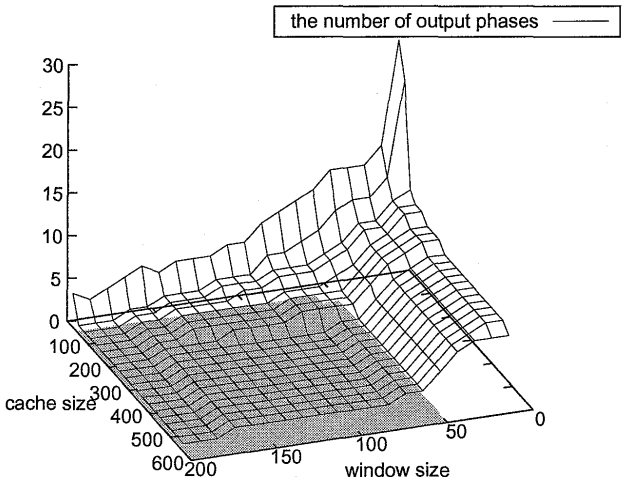

図 2 パラメータ値に対する出力開始点の個数

キャッシュサイズ・ウィンドウサイズの値が小さくなるにつれ，分割数が増加している．特に，2 つのパラメータどちらかの値を極端に小さくした時，分割数が一気に増大する．

また，x 軸に実行履歴上の時刻，y 軸にキャッシュサイズ・ウィンドウサイズの値をとり，それぞれもう片方のパラメータの値を固定した際に出力する開始点の分布を図 3 に示す．

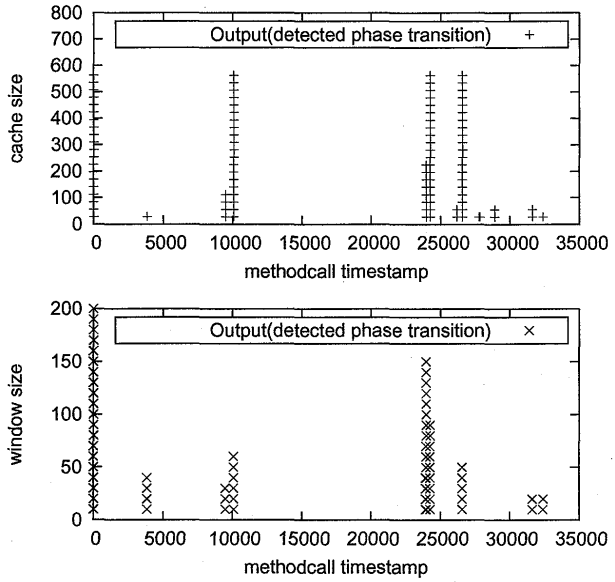

図 3 各パラメータ値よる分割フェイズの変化

キャッシュサイズ・ウィンドウサイズをそれぞれ変化させた場合，出力結果はその変化に伴って統合・細分化される形で変化している．

4.3.2 適合率

分割数ごとの平均適合率を図 4（Precision(All)）に示す．

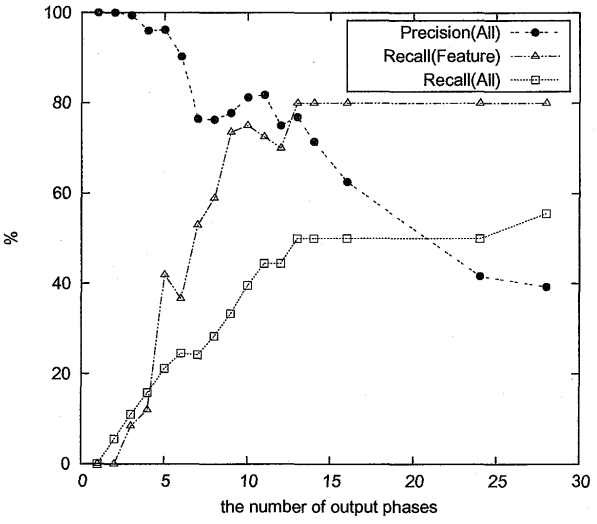

図 4 分割数と適合率・再現率

キャッシュサイズ・ウィンドウサイズどちらかのパラメータが極端に小さい時，開発者の指定したフェイズ開始時刻と一致しない誤検出点が増大して適合率が大幅に下がっている．これは，パラメータの値が小さいほど分割数が増大することから，出力結果が開発者の指定したフェイズ開始点の数を大幅に上回った結果も含まれるが，片方のパラメータだけが小さい場合には分割数が少ないながらも誤検出点が生じている場合があった．

一方で，適合率が 100% となる出力も，出力分割数が 1 から

8の間の多数の場合で存在した（図2において，値域をグレーで着色している範囲）．しかし，分割数が少ない場合にも表れやすい，特定の誤検出点も存在し，これが各分割数ごとの平均適合率を下げる要因になっていた．

4.3.3 再現率

出力された開始点の個数ごとの平均再現率を図4（Recall(All)）に示す．

分割数が増えるほど，再現率が向上する．これは，適合率とトレードオフの関係にある．

しかし，分割数を最大にしても，再現率は100%に届かなかった．これは，検出することが困難なフェイズが存在したためである．人間から見て重要な1つの処理であっても，実行履歴上での規模（動作したオブジェクト集合の要素数や，実行区間の長さ）が小さいフェイズ，たとえばプログラムの終了処理は，本手法では検出できなかった．

機能（Feature）単位で指定されたフェイズに対する再現率（図4（Recall(Feature)）と，各機能を構成する細粒度のフェイズに対する再現率（図4（Recall(All)））を比較した時，機能単位のフェイズに対するの再現率が先に上昇した．これは本手法で，機能（Feature）に対応するフェイズによる分割が優先的に行われやすいことを表す．

5. 考察

パラメータとしてあたえたキャッシュサイズ・ウィンドウサイズの値と出力の分割数が連動し，かつ粗粒度のフェイズを細分化していく形で出力分割点が増加していくことから，2つのパラメータを用いることで，認識したいフェイズの粒度を制御することができると言える．そのためには，求めるフェイズに対応したパラメータ値の算出式を定める必要があり，それは今後の課題である．

また，粗粒度のフェイズに対応する開始点が優先的に取得でき，かつ，分割数が小さいときは適合率を100 %に維持できることから，本手法は機能（Feature）単位のフェイズに対応した分割に適していると考えられる．

更に，分割数が小さい場合も表れやすい特定の誤検出点が存在したことと，粗粒度のフェイズに対応する開始点であっても，履歴上の規模や位置によっては検出されない場合があることから，本手法で分割されるフェイズの表す単位と，人間の考えるフェイズの単位は完全には一致しないことが判明した．この違いの原因については，今後，調査を行う予定である．

6. まとめと今後の課題

本研究では，与えられた実行時のメソッド呼び出し系列から，登場するオブジェクトとコールスタックの情報を用いて，各フェイズが開始する点を検出することで実行履歴を自動的に分割する手法を提案し，適用実験によってその妥当性を示した．

今後の課題としては，求める粒度に応じた出力を取得するためのパラメータ値の算出式を決定することが挙げられる．また，本手法で出力されるフェイズの意味する単位を定義することも必要である．

謝辞　本研究は，日本学術振興会科学研究費補助金若手研究（スタートアップ）（課題番号:19800021）の助成を得た．

文献

[1] Cornelissen, B., van Deursen, A., Moonen, L. and Zaidman, A.: Visualizing Testsuites to Aid in Software Understanding. Proc. of CSMR, pp.213-222, 2007.

[2] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code. IEEE Computer, Vol.29, No.3, pp.210-224, 2003.

[3] Hamou-Lhadj, A. and Lethbridge, T.: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System, Proc. of ICSE, pp.181-190, 2006.

[4] Kollmann, R. and Gogolla, M.: Capturing Dynamic Program Behavior with UML Collaboration Diagrams. Proc. of CSMR, pp.58-67, 2001

[5] LaToza, T. D., Venolia, G. and DeLine, R.: Maintaining Mental Models: A Study of Developer Work Habits. Proc. of ICSE, pp.492-501, Shanghai, China, 2006.

[6] Lejiter, M., Meyers, S. and Reiss, S. P.: Support for Maintaining Object-Oriented Programs. IEEE TSE, Vol.18, No.12, pp.1045-1052, 1992.

[7] Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects. Communications of the ACM, Vol.26, No.6, pp.419-429, June 1983.

[8] Object Management Group, UML 2.0 Infrastructure Specification. www.omg.org, 2003.

[9] 大平 直宏, 谷口 考治, 石尾 隆, 神谷 年洋, 楠本 真二, 井上 克郎: 動作オブジェクト群の変化に着目したオブジェクト指向プログラムの実行履歴分割手法. 電子情報通信学会論文誌 D-I, Vol.J88-D-I, No.12, pp.1810-1812, 2005.

[10] Pauw, W. D., Jensen, E., Mitchell, N. Sevitsky, G., Vlissides, J. M. and Yang, J.: Visualizing the Execution of Java Programs. Revised Lectures on Software Visualization, International Seminar, pp.151-162, 2002.

[11] Reiss, S. P. and Renieris, M.: Encoding Program Executions. Proc. of ICSE, pp.221-230, 2001.

[12] Richner, T. and Ducasse, S.: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. Proc. of ICMS, pp.34-43, 2002.

[13] Rountev, A., Volgin, O. and Reddoch, M.: Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams. Proc. of PASTE 2005, pp.96-102, 2005.

[14] Rountev, A. and Connell, B. H.: Object Naming Analysis for Reverse-Engineered Sequence Diagrams. Proc. of ICSE, pp. 254–261, 2005.

[15] Spinellis, D.: Abstraction and Variation. IEEE Software, Vol.24, No.5, pp.24-25, 2007.

[16] 谷口 考治, 石尾 隆, 神谷 年洋, 楠本 真二, 井上 克郎: プログラム実行履歴からの簡潔なシーケンス図の生成手法, コンピュータソフトウェア, Vol.24, No.3, pp.153-169, 2007.

[17] Tonella, P. and Potrich, A.: Reverse Engineering of the Interaction Diagrams from C++ Code. Proc. of ICSM, pp.159-168, 2003.

[18] Ungar, D.: Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. Proc. of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. pp.157-167, 1984.

[19] Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs. IEEE TES, Vol.18, No.12, pp.1038-1044, 1992.