修士学位論文

題目

オブジェクトの協調動作を用いた
オブジェクト指向プログラム実行履歴分割手法

指導教員
井上 克郎 教授

報告者
渡邊 結

平成 20 年 2 月 8 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 19 年度 修士学位論文

オブジェクトの協調動作を用いた
オブジェクト指向プログラム実行履歴分割手法

渡邊 結

内容梗概

オブジェクト指向プログラムは，動的束縛や例外処理など，実行時の状況によってその振る舞いが決定される制御要素を含んでおり，実際にプログラムがどのような処理を実行するのか，ソースコードの記述だけではプログラムの動作を理解することが困難となっている．そのため，テストやデバッグ段階で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から，オブジェクトの協調動作を可視化することが有効である．可視化のための手法・ツールは多く提案されているが，その際に用いられるプログラムの実行履歴には複数の機能の実行が重複込みで含まれ，サイズが巨大化するという問題が存在する．このため，可視化手法で示された図は巨大化し，また図中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を開発者が読解する必要が生じる．そこで本研究では，オブジェクト指向プログラムから取得した実行履歴を，特定の機能を実行した時間軸上の区間（フェイズ）の列へと自動的に分割する手法を提案する．提案手法は，1 つのフェイズ内で協調動作するオブジェクトの集合の変化を LRU キャッシュアルゴリズムを用いて検出し，新たなフェイズの開始位置として認識する．これは，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を生成しながら実現され，その機能の実行が終了する際にそれら多数の作業用オブジェクトを破棄するという性質に基づいたものである．本研究では，提案手法を 2 つのシステムに対して適用し，対象のシステムや実行履歴の内容に関する詳細な知識を用いることなくフェイズが検出できることを示した．

主な用語

フェイズ検出 (Phase Detection)
動的解析 (Dynamic Analysis)
実行履歴（Execution Trace)
プログラム理解 (Program Understanding)

目 次

1　まえがき

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，また継承や多態性を利用して，処理の階層化や共通化を行う．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によって呼び出しうる複数のメソッド定義を，開発者が手作業で発見，調査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 [7, 19, 24]．これに対し，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・可視化するアプローチが有効であると主張されている [15]．

オブジェクト指向プログラム実行履歴とは，その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．実行履歴を可視化する手法の一例としては，オブジェクトの ID よよびメソッド名を用いてシーケンス図を生成する手法がある [20]．ただし，実行履歴を単純にシーケンス図などの形式で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ないため，パターン検出を用いた圧縮処理 [13] や，ループや再帰呼び出しの検出 [20]，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 [12] など，可視化された図の可読性を向上させるための様々な手法が適用されている．

しかし，たとえ可視化された実行履歴が人間の読解に適したサイズにまで縮約されていても，1 つの実行履歴中には，複数の機能の実行が連続して記録されているため，その図中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，開発者が自ら識別しなければならない．

そこで本研究では，オブジェクト指向プログラムから取得した実行履歴を，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．特定の機能を実行したフェイズとは，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に対応した，実行履歴中のイベント列である．このようなフェイズを自動的に識別することで，開発者は実行履歴中から特定の機能に対応する部分の詳細だけを選択して調査することが可能となる．

フェイズを認識する具体的な方法としては，実行時に動作するオブジェクト集合の変化に着目する [11]．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，ある一定区間で多数の新しいオブジェクトが出現したとき，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，実行履歴上を過去へ遡って探索し，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．

提案手法の有効性を確認するために，企業で開発された 2 つのシステムに対する適用実験

を行った．適用実験では，設計ドキュメントに書かれたシナリオを実行して取得した実行履歴を用いて，提案手法によって出力されるフェイズと，開発者が手動で識別したフェイズとを比較した．また，同一システム上で異なるシナリオを実行した際の実行履歴や，実装の異なるシステム上で同一のシナリオを実行した際の実行履歴に対して，本手法のフェイズ分割結果がどのように変化するかを調査した．その結果，提案手法が実行シナリオや実装の変更の影響を受けにくく，開発者の判断に対する高い適合率が得られることを確認した．

以降，2 章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，3 章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 章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述べ，最後に 5 章でまとめと今後の課題について述べる．

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，また継承や多態性を利用して，処理の階層化や共通化を行う．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によって呼び出しうる複数のメソッド定義を，開発者が手作業で発見，調査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 [7, 19, 24]．

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 [10]．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 [15]．

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 [6]．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 [20]．

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可視化する方法が研究されている．このような可視化手法は，ソースコードを用いる静的解析と，実行履歴を用いる動的解析に大別できる．

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 [21]，制御フローグラフからシーケンス図を生成する手法 [16] が提案されている．また，これらの手法で生成した図中に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 [17] も提案されている．

ソースコードの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．これに対し，動的解析と呼ばれる，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 [20]．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 [13] や，ループや再帰呼び出しの検出 [20]，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 [12] 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 [12]．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，実行時に動作するオブジェクト集合の変化に着目する [11]．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 [8]．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 [22]．

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，それらの機能（Feature）の詳細な実行シナリオにおける各ステップである．識別されたフェイズが，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に対応したものとなることを目標とする．

本稿では，フェイズの粒度を明確に区別する時，以下に示す 2 種類の表記を用いる．

- **機能フェイズ**(*feature-level phase*) とは，ソフトウェアに対する要求を実現する 1 つの機能（Feature）の実行に対応するフェイズを意味する．あるユースケースシナリオを実行したときに記録される実行履歴は，シナリオ上の 1 つのステップとして表される 1 つの機能に対応する機能フェイズがシナリオ上で示された順序で並んだものとなる．
- **サブフェイズ**(*minor phase*) とは，各機能（Feature）詳細な実行シナリオの各ステップに対応する実行履歴上の区間を意味する．実行履歴から 1 つの機能フェイズを抽出したとき，それは 1 つ以上のサブフェイズからなる列である．

本研究において検出しようとしている上記のフェイズの定義は，Reiss が示したフェイズの定義 [14] と，Salah らによるイベント列としての feature の定義 [18] に基づいている．

フェイズ検出には，これ以外に，たとえば次のような定義も存在する．Wang は構文構造に基づくフェイズ検出を用いた階層的ダイナミックスライシング [23] を提案している．ここで提案された構文情報に基づくフェイズは階層関係を持ち，開発者が実行履歴をから障害の詳細を追う場合などに役立つ．しかし，Wang の定義するフェイズはユースケースに記述される機能との対応を保証しない．

プログラムの最適化の分野においても，フェイズという概念が用いられる．しかし，プログラムパフォーマンスの最適化手法におけるフェイズとは，実行を一定の間隔（たとえば，実行時間上で 10*milliseconds* [14] ごと）で分割したものを指すのが一般的である．動的にフェイズ間の移行を検出するため固定長でないフェイズを用いる手法 [9] も存在するが，いずれもソフトウェアの機能に依存するものではなく，本研究の検出するフェイズとは異なるものである．

これらと比べた場合，我々の用いるフェイズの定義は，以下の特徴を持つ．

- 機能（feature）あるいはそれを実現するための各ステップに対応している．
- 時系列上の連続したイベント列であり，長さには特に制限がない．

我々の手法が検出するフェイズを用いると，たとえば「データを編集したときにプログラムがクラッシュした」といった手がかりに対して，実行履歴のうちその機能に対応する一部だけを，開発者が限定して調査することができるようになることが期待される．

```
   :
@1 19 void LoginForm(5).<init>(){
@1 }
@1 20 boolean index_jsp(3)._jspx_meth_html_text_0(Tag,PageContext){
@1 21 String LoginForm(5).getShimeiNo(){
@1 }
@1 }
@1 22 boolean index_jsp(3)._jspx_meth_html_password_0(Tag,PageContext){
@1 23 String LoginForm(5).getPassword(){
   :
```

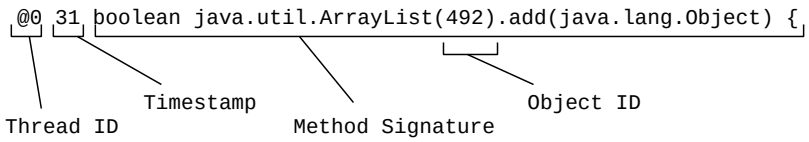

図 1: Java プログラムから取得した実行履歴（部分）

3 提案手法

プログラムの各機能は，その実行のために多数のオブジェクトを生成し，終了時にはそのほとんどを破棄するという性質 [8, 22] を持つ．提案手法は，この性質を利用して，動作しているオブジェクトの集合を監視することでフェイズの変化を検出する．具体的には，Least-Recently-Used キャッシュを用いて,「最近動作したオブジェクトの集合」の監視を行う．

提案手法は，実行履歴上から取得する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，分割さ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 実行履歴

提案手法が分割対象とする実行履歴は，実行されたメソッド呼び出しイベントの列からなるものである．履歴中のメソッド呼び出しイベントはそれぞれ，以下に示す 4 種類の情報を保持しているものとする．

イベント時刻(*timestamp*) メソッド呼び出しが行われた時刻（履歴上における順番）を表す．一般に，各イベントが実行履歴上に登場する順に整数値が割り当てられる．同一履歴中に存在するすべてのイベントは，それぞれ異なるタイムスタンプを割り振られる．

呼び出し先オブジェクト(*calleeID*) メソッド呼び出しイベントにおいて，呼び出された側

のオブジェクトを表す．実行履歴上に登場するすべてのオブジェクトは，それぞれ独立した識別子を持つものとする．

呼び出し元オブジェクト(*callerID*) メソッド呼び出しイベントにおいて，呼び出した側のオブジェクトを表す．

コールスタックの深さ(*callstack*) そのメソッド呼び出しイベントが実行された時点での，コールスタックの深さを表す．

具体例として，図1 に Java プログラムから取得した実行履歴の一部を示す．これは，Java プログラムの実行履歴を取得するツール Amida Profiler [20] を用いて記録されたものである．このツールは Java Virtual Machine Profiler Interface (JVMTI) を用いて Java プログラム実行時のメソッド呼び出しイベント及びリターンイベントを監視し，その結果を図1 に示す形式で出力する．

3.2 アルゴリズム

提案手法のアルゴリズムを図2 に示す．

提案手法への入力は，実行時のメソッド呼び出しイベント系列 E である．イベント系列 E は，イベント時刻 t におけるイベントオブジェクト e_t によって構成される集合である．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，$e_t.callstack$ 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，実行履歴上のフェイズ開始点に相当するイベント時刻の列である．

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 [11]」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，実行履歴をフェイズ単位で分割する．

以下，各段階の具体的なアルゴリズムについて述べる．

3.2.1 フェイズ移行の検出

図2 において，$frequency(t, w)$ 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 $frequency(t, w)$ 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 内の要素にヒットしなければその新しいオブジェクト ID を新たに追加し，その際キャッシュC に新しい要素を追加する空きがない場合は最も参照時刻が古い要素を破棄する．

図 2(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 2(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 $frequency$ を次式によって計算する.

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

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 $frequency(t, w)$ 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 $frequency(t, w)$ の値が 1 へ近づく．すなわち，更新頻度 $frequency(t, w)$ の値が**閾値**$threshold$ の値を超えた時，フェイズの移行が生じていると判断できる.

3.2.2 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 $frequency(t, w)$ 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，すなわ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．したがって，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 $frequency(t, w)$ 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 $frequency(t, w)$ が**閾値** $threshold$ を超えた時刻」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 2 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．

ここで，m は過去何回分のメソッド呼び出しをフェイズ開始点の候補として調査するかを指定するパラメータであり，以後**スコープサイズ** と呼ぶ．

3.2.3 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，ここまでの記述で登場した 4 つのパラメータ「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」を使用する．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．

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，値は整数値をとり最小で 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．

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，値は整数値をとり最小で 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．

閾値 $threshold$　$frequency(t, w)$ の値からフェイズの移行を検知する閾値であり，値は実数値をとり最小で 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，$frequency(t, w)$ 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 $frequency(t, w)$ 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 $frequency(t, w)$ だけを切り分けるような閾値を設定することは難しい．

そこで 4 章の適用実験では，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．

スコープサイズ m　過去何回分のメソッド呼び出しをフェイズ開始点の候補として調査するかを指定するパラメータであり，値の範囲はウィンドウサイズと同様に整数値をとり最小で 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 $frequency(t, w)$ 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する可能性や，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 章の適用実験では，スコープサイズも固定値を用いた．

3.3　計算量とコスト

提案手法の計算量は，入力する実行履歴のサイズ（メソッド呼び出しイベント数）を n としたとき，$O(mn)$ となる．m は上述したパラメータの一つである**スコープサイズ** の値である．また，本手法は計算中にパラメータ**キャッシュサイズ** c で指定される大きさの固定長キャッシュと，パラメータ**ウィンドウサイズ** と**スコープサイズ** の値に応じた $\max(m, w)$ 回分のメソッドコールイベント情報をメモリ上に保持している必要がある．しかし，これら 3

つのパラメータ c, w, m の値は，履歴長 n に比べて十分小さい．そのため，本手法のアルゴリズムは入力する実行履歴のサイズにほぼ比例したコストで計算可能である．

procedure DetectPhases(

in $E = [e_1, e_2, \cdots e_{last}]$;

in $c, w, m : integer$; $threshold : double$;

out $P : set\ \ of\ \ timestamp$

(1) C = new LRUCache(c);$P \leftarrow \phi$

(2) **for** t in $[1 \ldots last]$

(3) $updated[t] = update(C, e_t.caller, e_t.callee)$

(4) **if** $frequency(t, w) \geq threshold$

(5) $P \leftarrow IdentifyPhaseHead(t, m)$

(6) **end if**

(7) **end for**

function update(**in** C, callerID, calleeID): integer

1) $b1 \leftarrow C.update(callerID)$ – b1, b2 = true if

2) $b2 \leftarrow C.update(calleeID)$ – C did not contain the ID

3) **if** $b1 \vee b2$ **then return** 1 **else return** 0

function IdentifyPhaseHead(**in** t, m) : integer

1) $min = x = \max(t - m + 1, 1)$

2) **while** $x \leq t$

3) **if** $e_{min}.callstack \geq e_x.callstack$ **then** $min = x$

4) $inc(x)$

5) **end while**

6) **return** min

図 2: 実行履歴分割手法のアルゴリズム

4　適用実験

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

4.1　実験方法

実験対象は下記２つの Java プログラムである．

- **ツール管理システム**は，実際にシステム開発企業が開発し，社内で運用している業務用 web アプリケーションである．このシステムは，サーバ制御用の１つのスレッドと，HTTP リクエスト処理用の３つのスレッドからなるマルチスレッド方式で実装されている．ソースコードの規模は行数にして約 37,000 行である．ユーザーインターフェイスは JSP を用いており，実行履歴上には各 JSP 上の処理と JSP 間遷移時のデータの受け渡し及びビジネスロジッククラスの処理が記録される．実験では，表 1 に示す 4 つのユースケースシナリオ（T-1, T-2, T-3, T-4）それぞれの実行履歴を取得し，本手法を適用した．4 つのユースケースシナリオはそれぞれ実行された機能や実行順序が異なる．「Login」のように複数のシナリオに含まれる機能も存在するが，該当する機能フェイズの詳細な内容は，実行履歴ごとに異なる．

- **書籍管理システム**は，システム開発企業が社内研修用に設計し，演習で実装されるソフトウェアである．実験では，同一の設計を用いて異なる実装が行われた５つのプログラムを対象とした．5 つのプログラムそれぞれで 表１に示す共通のユースケースシナリオ（L-1,2,3,4,5）の実行履歴を取得し，本手法を適用した．シナリオが共通であるため，5 つの実行履歴は共通の機能フェイズで構成される．しかし，それぞれプログラムの実装が異なるため，同じ機能フェイズを構成するサブフェイズが異なる場合もある．

上記２つのシステムでそれぞれ表１に示すユースケースシナリオを実行し，Amida Profiler [20] を用いて実行履歴を取得した．

取得した各実行履歴を各システムの開発者に手動で解析していただき，実行したユースケースシナリオに対応する各機能フェイズの正確な位置（フェイズの開始点）を決定した．更に，実行履歴上の各 機能フェイズを，その機能を構成する各実行ステップに対応する複数の サブフェイズに分割していただいた．取得した実行履歴から読み取れる情報とそれぞ

表 1: 実行履歴を取得したユースケースシナリオ（機能フェイズ）

ID	*Scenario*
T-1	Login → Listing tools → Maintenance view of a tool → Updating the tool information → Logout
T-2	Login → Listing tools → Maintenance view of a tool → Updating the tool information → Cancelling the edit → Logout
T-3	Login → Listing tools → Maintenance view of a tool → Logout
T-4	Login → Searching tools → Maintenance view of a tool → Shutdown
L-1,2,3,4,5	Login → Showing mylist → Adding a new book → Registering a new book → Showing booklist → Searching books → Borrowing a book → Showing the detail of a book → Returning a book → Marking a book → Unmarking a book → Showing browwinglist → Logout → Login as a new user → Logout

れのシステムの開発者の認識したフェイズ数を表 2 に示す．*#events* は実行履歴上に記録されたイベント数（メソッド呼び出し回数）であり，実行履歴の長さを表す．*#objects* は実行履歴上に登場したオブジェクトの総数を表す．*#feature-level phases* は実行履歴を構成する機能フェイズの数を表し，これは実行したユースケースシナリオ上のステップ（表 1）の個数と一致する．*#minor phases* は実行履歴を構成する サブフェイズの数を表し，これは実行履歴中の各 機能フェイズを構成する サブフェイズの数の合計である．

その上で，各実行履歴に対してパラメータを変化させながら本手法を適用し，出力されたフェイズと開発者が認識するフェイズとの比較を行った．

4 つのパラメータのうち，**閾値** と **スコープサイズ**はそれぞれ $threshold = 0.1$ ， $m = 700$ という固定値を用いた．これは，残り 2 つのパラメータと比較して，提案手法の出力に与える影響がごく小さかったためである．

残り 2 つのパラメータ **キャッシュサイズ** ，**ウィンドウサイズ**は，提案手法の出力結果に与える影響が大きいため，各システムについて 10 ずつ変化させながら複数の組み合わせで手法を適用し，出力結果を収集した．ツール管理システムの各実行履歴に対しては，**キャッシュサイズ** c を 10 から 600 まで 10 刻みで，**ウィンドウサイズ** w を 10 から 200 まで 10 刻みで用いた．図書管理システムの各実行履歴に対しては，**キャッシュサイズ** c を 10 から 350 まで 10 刻みで，**ウィンドウサイズ** w を 10 から 300 まで 10 刻みで用いた．**キャッ**

表 2: 各実行履歴の詳細

System	*ID*	*#events*	*#objects*	*#feature-level phases*	*#minor phases*
ツール管理システム	T-1	32416	546	5	18
	T-2	30494	524	6	19
	T-3	26603	438	4	14
	T-4	15909	237	3	10
書籍管理システム	L-1	3573	261	15	52
	L-2	3371	272	15	51
	L-3	3797	286	15	51
	L-4	3862	300	15	51
	L-5	4506	341	15	64

シュサイズ c の値は，各システムの実行履歴それぞれに登場するオブジェクト数の最大値を超える値までとした．**ウィンドウサイズ** w を，各実行履歴のイベント数ではなくそれぞれ 200, 300 までとしたのは，それ以上の値で手法を適用しても出力結果が変化しなかったためである．ツール管理システムの 4 つの実行履歴それぞれにつき 1200 通り，図書管理システムの 5 つの実行履歴それぞれにつき 1050 通りのパラメータ設定で，提案手法による実行履歴分割を行った．

なお，この実験は Amida Profiler の出力をメソッド呼び出しイベント系列に変換し，本手法を適用しフェイズ開始点系列を出力するまでを，Perl スクリプトを作成して実現した．演算は Xeon 3.0 Ghz の CPU を備えた workstation 上で行い，上述したすべて（計 10050 通り）の結果を出力し終えるまでに 5 分弱の時間がかかった．

提案手法では，入力された各実行履歴上からフェイズを検出し，その個数と開始位置を出力する．本手法の出力結果と，開発者が判断したフェイズとを比較する際の評価基準として，以下に示す適合率と再現率を用いた．

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

提案手法が出力したフェイズの開始点の集合を P，開発者が認識したフェイズの開始点の集合を $Manual$ とし，また集合 S の要素数を $|S|$ と表記している．

4.2 実験結果

4.2.1 パラメータと出力フェイズ数

キャッシュサイズと**ウィンドウサイズ**の 2 つのパラメータの値の変化に対して，本手法が出力するフェイズの個数の変化の一例を図 3 に示す.

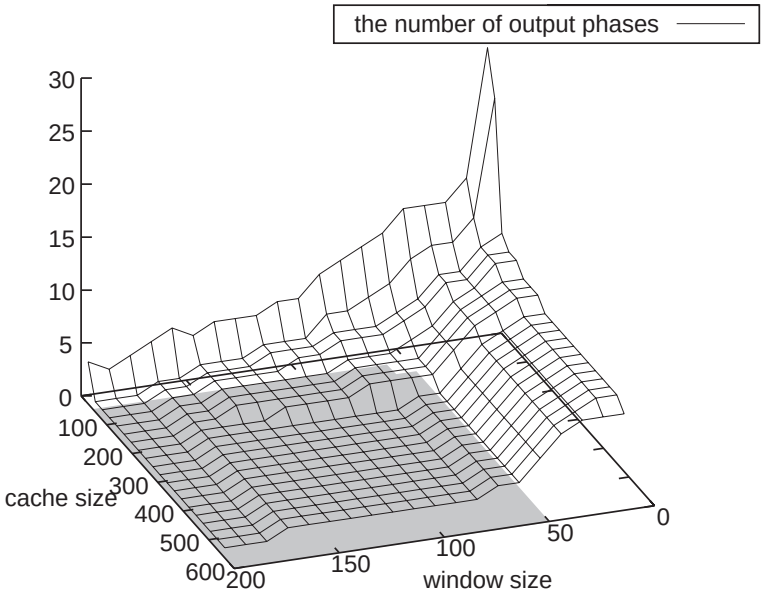

図 3: 履歴 T-1 における，各パラメータ値に対する本手法の出力フェイズ数

キャッシュサイズ または**ウィンドウサイズ** の値が小さいほど，本手法が出力するフェイズ数が増加している．履歴 T-1 を含むすべての出力結果から，**キャッシュサイズ** および**ウィンドウサイズ** の値と，本手法が出力するフェイズ数には相関があることが読み取れた．また，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傾向が見られた.

x 軸に実行履歴上の時刻，y 軸に**キャッシュサイズ** または**ウィンドウサイズ** の値をとり，それぞれ他方のパラメータの値を固定した際に出力するフェイズ開始点の分布を図 4 に示す．上側の図は**ウィンドウサイズ** を $w = 50$ に固定した際の出力を表す．図中にプロットされた各点は，本手法で出力されたフェイズの開始点を表している．(x, y) に点がプロットされている時，これは **キャッシュサイズ** $c = y$，**ウィンドウサイズ** $w = 50$ のパラメータで出力されたフェイズの 1 つが，履歴上で x 番目のイベントから開始していることを表す．同様に，下側の図は**キャッシュサイズ** を $c = 100$ に固定した際の出力を表している.

図中に網掛けの縦棒で覆われている点は，本手法の出力のうち，開発者の認識する機能フェイズの開始点に正確に一致していたものを表す．同じく，網掛けの四角で囲まれている

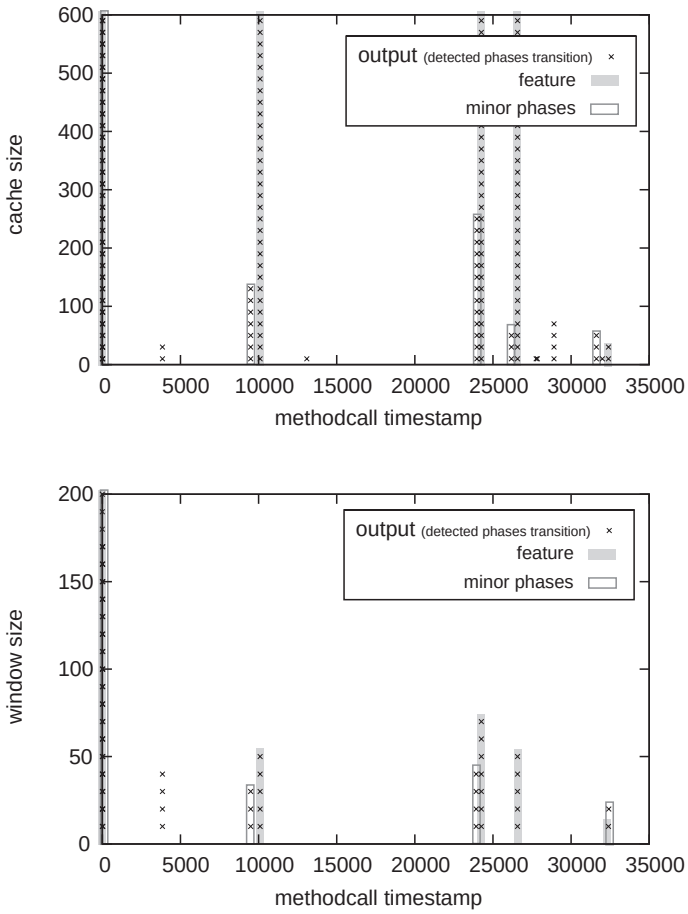

図 4: 履歴 T-1 における出力フェイズの開始点.
（上図）ウィンドウサイズを固定した場合 ($w = 50$),
（下図）キャッシュサイズを固定した場合 ($c = 300$).

点は，本手法の出力のうち，開発者の認識する機能フェイズの開始点ではないが，開発者の認識するサブフェイズの開始点に正確に一致していたものを表している．

パラメータのうち 1 つだけを変化させた時，本手法で出力されるフェイズの数はそれに伴い増減するが，このとき出力されるフェイズは統合もしくは細分化される形で変化している．つまり，出力されたフェイズが n 個であるパラメータの組み合わせに対して，片方のパラメータを出力が $n+1$ 個になるまで小さくした場合，先に出力された n 個のフェイズのうち 1 つを 2 つに分割するかたちで出力フェイズが増加する．この時，先に出力された n 個のフェイズの開始点はそのまま保持されているということである．LRU キャッシュが更新されるイベント時刻および各時刻での $frequently$ の値の変化は，パラメータが異なれば全く異なる位置に変化してしまう．しかし本手法では，パラメータの値を小さくしても，フェイズ

の開始点を安定して同じ時刻で検出し続けている．

ただし，2 つのパラメータどちらかの値を極端に小さくしていった時はこの限りではなく，出力フェイズ数が爆発的に増えると共に，これまでに検出していたフェイズの開始点を検出しなくなる場合がある．

4.2.2 適合率と再現率

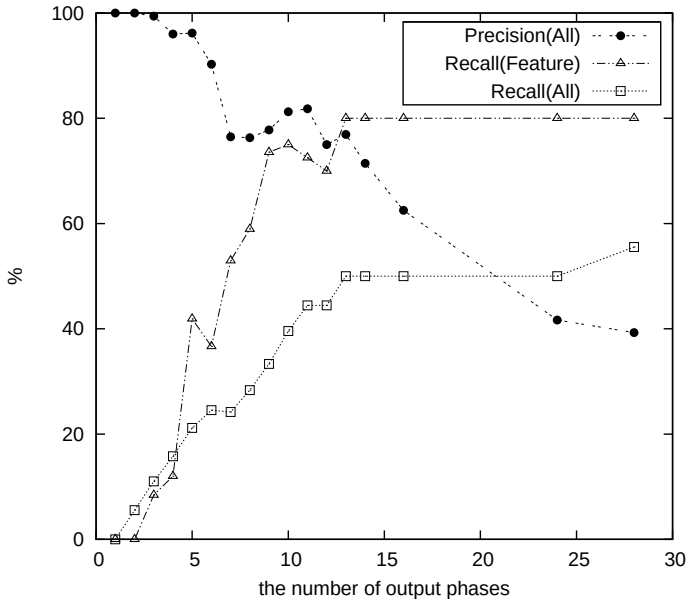

図 5: 履歴 T-1 における，出力フェイズ数ごとの平均適合率および平均再現率

履歴 T-1 における 1200 通りの出力結果を，出力フェイズ数ごとに分類し，それぞれの適合率と再現率の平均を図 5 に示す．X 軸は出力されたフェイズの個数を表しており，Y 軸は出力フェイズ数が同数であったパラメータ組の平均適合率 *Precision(All)* ，平均再現率 *Recall(Feature)* （ただし，T-1 に含まれる 5 つの機能フェイズのみに対する再現率）・*Recall(All)* を表す．

出力フェイズ数が少ない場合に非常に高い平均適合率 *Precision(All)* を示している．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 3 において，値域を網掛け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た．また，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多くの誤検出点が生じている場合があり，これらが平均適合率を引き下げる要因になっていた．

分割数が増えるほど，平均再現率 *Recall(All)* が向上する．これは，適合率とトレードオ

表 3: すべてのパラメータの組み合わせに対する，出力フェイズ数ごとの平均適合率・再現率

ツール管理システム

#phases	*Recall(Feature)*	*Recall(All)*	*Precision*
5	0.56	0.39	0.93
10	0.90	0.48	0.80

書籍管理システム

#phases	*Recall(Feature)*	*Recall(All)*	*Precision*
10	0.24	0.20	0.99
15	0.53	0.29	0.98
20	0.45	0.38	0.96

フの関係にある．

しかし，1200 通りのパラメータ組のうちどの組み合わせでも，出力の再現率は 100% に届かなかった．これは，本手法で検出することが困難なフェイズが存在したためである．例えば，機能フェイズに対する平均再現率 *Recall(Feature)* は最大で 80% だが，これは表 1 に示される T-1 の 5 つの機能フェイズのうち *Logout* に対応するフェイズが検出されなかったためである．*Logout* フェイズで実行されるメソッド数が特に少なく，このためフェイズ内で動作するオブジェクトの数も少ない．このように，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（フェイズ内で動作したオブジェクト集合の要素数や，実行区間の長さ（イベント数））が小さいフェイズは，本手法では検出できなかった．

ツール管理システムの 4 つの実行履歴それぞれに 1200 通りのパラメータを与えて分割した合計 4800 通りの出力のうち，出力フェイズの数が 5 個と 10 個であった組み合わせについて，それぞれ適合率・再現率の平均値を表 3 に示す．同様に，**書籍管理システム**の 5 つの実行履歴それぞれに 1050 通りのパラメータを与えて分割した合計 5250 通りの出力のうち，出力フェイズの数が 10 個，15 個，20 個であった組み合わせについて，それぞれ適合率・再現率の平均値を表 3 に示す．

例えば，**ツール管理システム** の実行履歴のいずれかを本手法で分割し，出力フェイズ数が 10 であった場合，平均で 8 つのフェイズが開発者の認識と一致するものであり，残り 2 つは誤検出である．また，その実行履歴を構成する機能フェイズの 90% は，検出した 10 個のフェイズのうち開発者の認識と一致する平均 8 つのフェイズのいずれかに該当し，加えてその 8 つのフェイズは実行履歴中のサブフェイズの約半分に一致する．

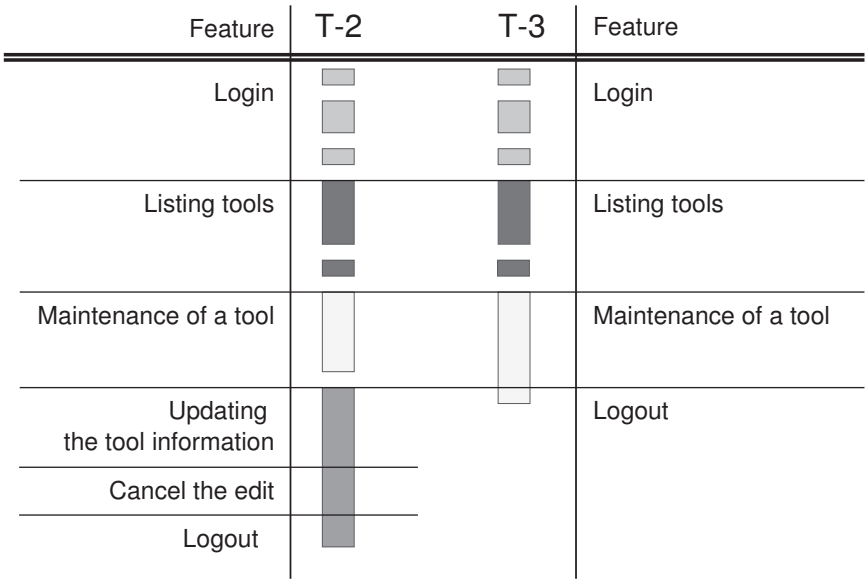

図 6: 実行履歴 T-2 と T-3 に同じパラメータを与えて本手法を適用した際の出力フェイズ

いずれのシステムにおいても，本手法は出力フェイズ数が少ない場合により高い平均適合率 *Precision(All)* を示す傾向がある．

機能フェイズに対する平均再現率 *Recall(Feature)* と，サブフェイズを含むすべてのフェイズに対する平均再現率 *Recall(All)* を比較した時，出力フェイズ数が増えるに従って，機能フェイズに対する再現率が先に上昇している．これは本手法で，機能フェイズが優先的に出力されやすいことを表す．しかし，出力フェイズ数が履歴中に含まれる機能フェイズの個数にほぼ等しい時（ツール管理システムで約 5 個，書籍管理システムで 15 個），機能フェイズに対する平均再現率 *Recall(Feature)* はどちらも 50% 程度である．すなわち，実行履歴中に含まれる機能フェイズの個数が判明しており，本手法によってその実行履歴を機能フェイズの個数に等しい数のフェイズに分割した場合，出力フェイズの開始点のうちおよそ半分は機能フェイズには対応していないただし，機能フェイズに対応していないフェイズであっても，その大多数は，開発者にとって意味のあるサブフェイズに対応する．

4.2.3 異なるユースケースシナリオ間の比較

ツール管理システムの実行履歴 T-2 と T-3 に対し，**キャッシュサイズ** $c = 100$，**ウィンドウサイズ** $w = 50$ の同じパラメータを与えて本手法を適用した場合の出力を比較した．

図 6 は，それぞれの実行履歴の機能フェイズに対する出力フェイズの位置を表す．図 6 は実行履歴中の時刻の流れを上から下へ縦方向にとっており，水平線は履歴中の各機能フェイズの境界位置を表す．ここで，2 つのシナリオの共通部分が隣り合うように図示しているが，

実行履歴 T-2 と T-3 上における各機能フェイズの境界となる点（実行履歴上での時刻）は必ずしも一致しないことに注意されたい．

図 6 において，網掛けで塗りつぶされた長方形が，検出された各フェイズのシナリオ上の位置を表している．網掛けの濃度は，各出力フェイズの開始点がシナリオ上のどの機能フェイズに含まれるかによって変化させている．長方形の個数が出力フェイズの数を表し，履歴 T-2 からは 7 個のフェイズを，T-3 からは 6 個のフェイズを検出している．このとき出力されたフェイズはすべて履歴中に含まれるいずれかのサブフェイズの境界で分割されており，誤検出は無かった．例えばここでは，本手法は双方の実行履歴の最初に現れる機能フェイズである「Login」フェイズを，3 つのフェイズに分割している．一つ目の出力フェイズは，機能フェイズ「Login」を構成する最初のサブフェイズである「ログイン画面表示」と一致する．二つ目の出力フェイズは，機能フェイズ「Login」を構成する二番目と三番目のサブフェイズ「ログイン処理」と「ユーザ情報の取得」に相当し，三つ目の出力フェイズは，機能フェイズ「Login」を構成する最後のサブフェイズである「トップ画面表示」に一致する．

2 つの実行履歴 T-2 と T-3 は，シナリオ上では同一の機能を実行している部分があり，その部分に該当する部分履歴は同一の機能フェイズから構成されており，またそれらの機能フェイズ一つ一つもそれぞれ同一のサブフェイズによって構成されている．しかし，シナリオ上で同一フェイズであっても，実行そのものはそれぞれ異なるオブジェクトと異なるメソッド呼び出しによって実現されている．

にも関わらず，図 6 からは異なる履歴からシナリオ上で同じフェイズを同じように検出していることが読み取れる．この結果から，本手法は異なるユースケースを用いた異なる実行履歴からであっても，同じパラメータを与えることで同じ機能を検出することができるといえる．

4.2.4 異なる実装間の比較

書籍管理システムの 5 つの実行履歴 T-1 から T-5 に対し，**キャッシュサイズ** $c = 150$, **ウィンドウサイズ** $w = 150$ の同じパラメータを与えて本手法を適用した場合の出力を比較した．

5 つの実行履歴 T-1,2,3,4,5 は，異なる 5 つの実装上で同じユースケースを実行したものであるので，実行履歴を構成する機能フェイズは共通であり，各機能フェイズはそれぞれ 2 個から 6 個のサブフェイズで構成されている．ただし，各実行履歴は実行したシステムの詳細な実装が異なるため，同じ機能フェイズが異なるサブフェイズによって構成されている場合がある．すなわち，5 つの実行履歴は同一のシナリオを実行したものであるが，実行時環境の差異によって履歴上のオブジェクトやメソッド呼び出しが異なるだけでなく，実装の違いによってサブフェイズのレベルで内容が異なっている．

図 7 は，5 つの履歴それぞれについて，機能フェイズに対する本手法の出力フェイズ位置を表す．図の表記様式は図 6 と同様である．出力フェイズ数は 9 個から 13 個であり，どのフェイズも各履歴中に含まれるいずれかのサブフェイズの境界で分割されており，誤検出は無かった．

本手法による出力フェイズが実行シナリオ上で占める位置を比較した時，5 つの実行履歴は非常に良く似た結果を示している．

例えば,「Returning a book」や「Marking a book」に対応する機能フェイズはどの履歴からも検出されているが,「Showing browwinglist」「Logout」に対応する機能フェイズはどの履歴からも検出されなかった．また，5 つの実行履歴はどれもシナリオ上で 5 番目に実行される機能フェイズ「Showing booklist」の開始点で分割されているが，その時点から始まる出力フェイズはどれも 6 番目に実行される「Searching books」と連続した 2 つの機能フェイズに対応するものになっている．

履歴 L-5 を除く 4 つの履歴においては，図 7 中で機能フェイズの境界を表す水平線と出力フェイズを表す長方形が少しずつずれて交わっている箇所が多く見られる．これは，該当する機能フェイズがいずれもごく小規模のサブフェイズ「初期化」から開始されており，本手法がこの「初期化」のサブフェイズではなく次に実行されたサブフェイズの開始点を検出してしまっているためである．これは,「初期化」サブフェイズが履歴上でごく小さい（メソッド呼び出し回数にして 10 回未満）であることと,「初期化」サブフェイズの開始点，すなわち親となる機能フェイズの開始点におけるコールスタックの深さが二番目に実行されるサブフェイズの開始点におけるコールスタックの深さが等しいことが原因である．

このように，同じシナリオを用いた実行履歴に対する本手法の適用は，実装が異なることによりサブフェイズレベルの差異は生じているものの，シナリオ上で共通する機能フェイズレベルでは，正しく検出できたフェイズ，検出できなかったフェイズ，細分化してしまうフェイズなど，機能フェイズごとにおおよそ似通った出力が得られている．

したがって，本手法は詳細な実装の差異による影響を受けにくいと考えられる．この出力の安定性は，例えばデバッグ作業でのバグ修正前と修正後など，実装が異なるシステムの実行を比較する必要がある際に非常に有効である．

4.3 考察

実験結果の妥当性 この実験は，本手法が実用に即したものであることを示すため，実際にシステム開発企業が用いているシステム上で，開発者の作成したユースケースシナリオを用いている．ただし，今回の実験の対象ドメインは企業アプリケーションのうち，データベース管理ソフトウェアに限られる．

また，使用した 2 つのシステムはマルチスレッドプログラムとして実装されているが，実

験に用いたユースケースシナリオはいずれもマルチスレッドによる並列処理や同期制御を必要とするものではなかった．このため，マルチスレッドプログラムの実行履歴に対する本手法の適用可能性は，別途検証が必要である．

更に，各実行履歴を取得する際，システム上で実行されたすべてのメソッド呼び出しイベントのうち，Java 標準ライブラリに対するイベントをフィルタリングしている．このため，用いた実行履歴上には Java 標準ライブラリに含まれるクラスのオブジェクトが登場しない．この理由は，第一に各フェイズがアプリケーション固有のオブジェクトによって特徴付けられ，Java 標準クラスのオブジェクトによる差異はより少ないと考えられるためである．第二に，実行履歴からシステムを理解する際に，Java 標準クラスのオブジェクトの動作を可視化する必要性が，Java 標準クラスのオブジェクトを除去することによる全体の可読性上昇より小さい場合が多いと考えられるためである．そして第三に，Java 標準クラスのオブジェクトを除去することによる結果への影響に比べて，Java 標準クラスのオブジェクトの動作を含めて実行履歴を記録するための実行時のオーバヘッドが大きすぎるためである．

フェイズと機能のマッピング　本手法の出力は，出力フェイズの開始点に当たる実行履歴上のイベント時刻系列である．そのため，開発者は実行履歴上の各出力フェイズがユースケースシナリオ上のどのステップに相当するのかを自力でマッピングしなければならない．このマッピング自体は，システムの実装に対する多少の知識が必要になってしまうものの，開発者にとってそれほど困難な作業ではない．しかし，たとえ同一のユースケースシナリオを用いた実行履歴であっても，その中身は実行時の環境や入力値の違いによって異なるものになる．そのため，開発者は分割したすべての実行履歴に対して，それぞれ機能とフェイズのマッピングを行わねばならず，そのコストは取得した履歴の数と分割したフェイズの数に比例して膨大になる．

この問題を解決するためには，本手法により分割した各フェイズに対して自動的に適切な識別名を割り当てるという手段が考えられる．同じ識別名を割り当てられたフェイズは同じ機能に相当するように名前を割り振ることで，開発者は同じ識別名を持つフェイズに対して一度だけ機能とのマッピングを行うだけで済み，コストが軽減される．

この問題に関連する研究分野としては，*feature location* や *traceability recovery* などが上げられる．Koschke らは各機能をその機能に固有のメソッドで特徴づける手法を提案しており [5]，また Rountev らはオブジェクトに対して変数名から適切な名前を与える手法を提案している [17]．これらの手法を応用し，本手法の出力したフェイズを登場するオブジェクトやメソッドによって特徴づけ，比較することで，適切な識別名を与えるという方法が考えられる．

実行履歴を自動で分割し，各フェイズに対応する機能が自動的にマッピングされれば，開

発者がプログラムのテストやデバッグを行う際に，現在何のテストを行っているのか，またプログラムが何を実行しているのかを Feature のレベルで把握することが容易になるため，特に有効であると考えられる．

可視化ツールとの連携　提案手法は，既存の実行履歴可視化ツールと連携し，実行履歴中から表示すべき区間をフェイズとして限定することで，可視化された図の可読性を向上させることが出来る．

図 8 は，実行履歴を簡潔なシーケンスダイアグラムとして表示するツールである Amida [20] を用いて，実験に用いた履歴 T-1 から本手法により検出した「Maintenance view of a tool」フェイズを可視化した際のスクリーンショットである．本手法を適用することで，対象システムに対する知識を用いることのないまま，シーケンスダイアグラムから他のフェイズに属する部分を除去することに成功している．ここで画面に表示されているシーケンスダイアグラムは，メソッド呼び出し回数にして実行履歴全体の 40%分であり，その分図中に描かれるオブジェクト数も減じるため，提示される図全体のサイズが大幅に縮小している．

このように，本手法によるフェイズ分割結果は，既存の他の可視化手法と組み合わせに効果的である．

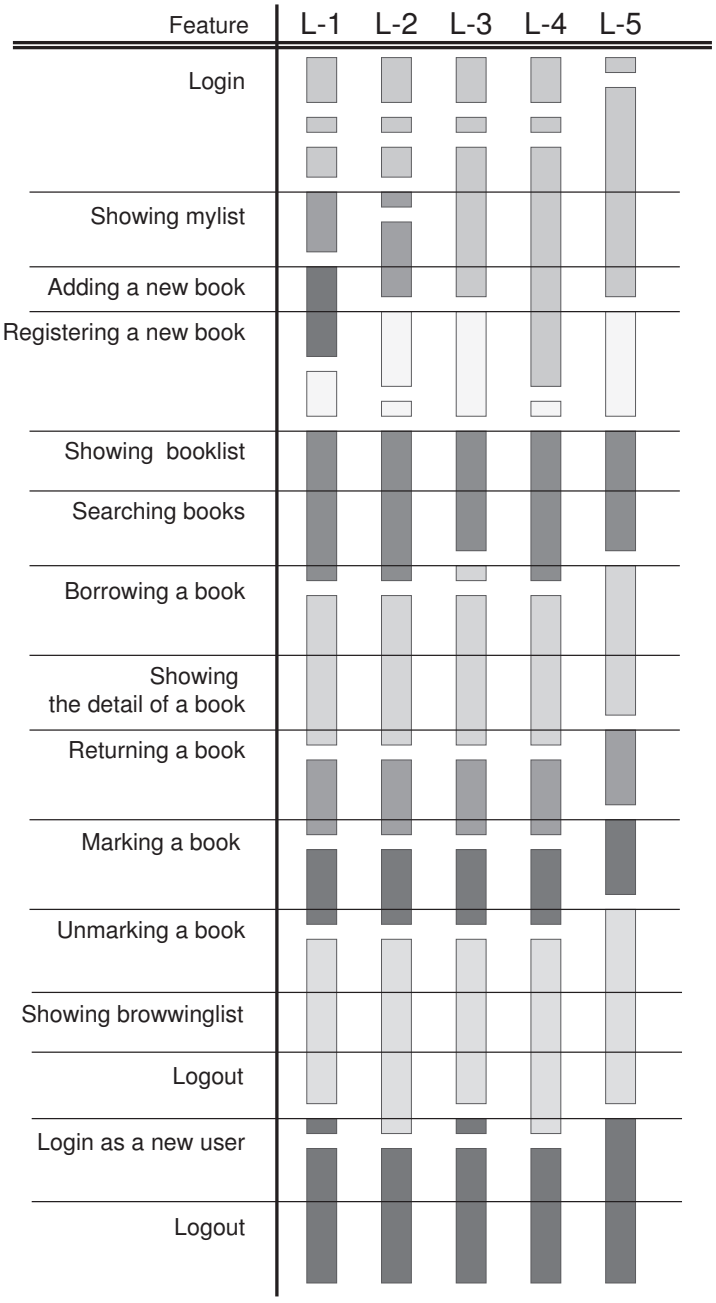

図 7: 書籍管理システムの 5 つの実行履歴に同じパラメータを与えて本手法を適用した際の出力フェイズ位置

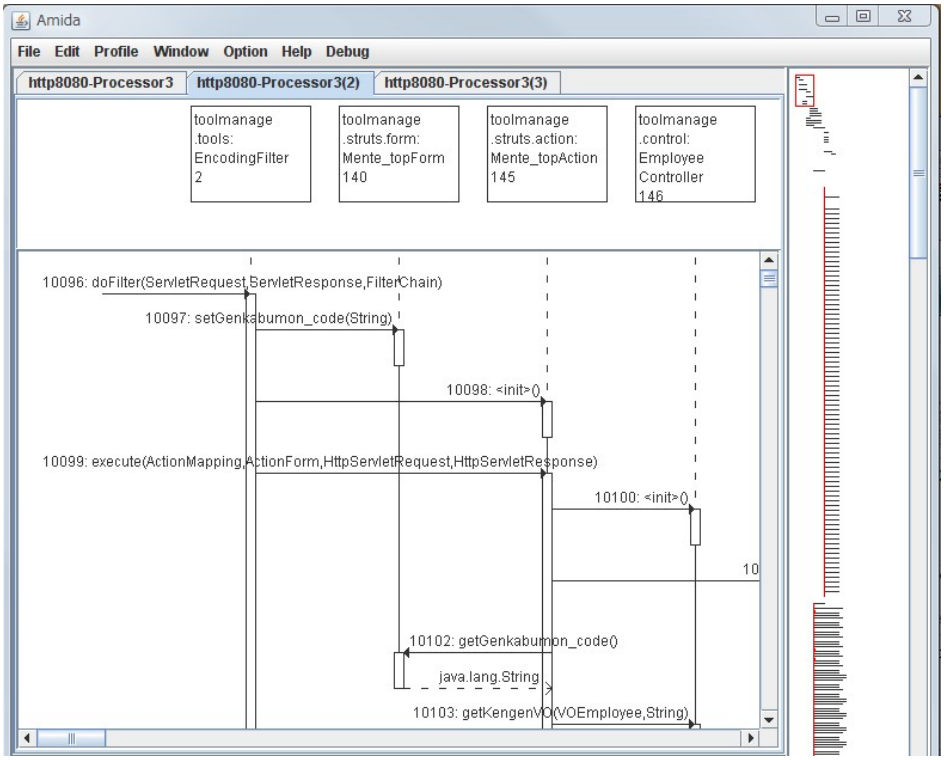

図 8: 本手法で検出した「Maintenance view of a tool」フェイズを実行履歴可視化ツール Amida で表示したスクリーンショット

5　まとめ

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．

本手法のアルゴリズムは，実行履歴上から読み取れる情報のみを用いており，また入力する実行履歴のサイズにほぼ比例した時間コストで計算可能である．実験の結果，十分高速な計算で高い適合率・再現率の出力を得ることができた．

パラメータとして与えたキャッシュサイズおよびウィンドウサイズの値と出力されるフェイズ数が連動し，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．ただしそのためには，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，機能フェイズに対応するフェイズ開始点を優先的に出力し，かつ，出力フェイズ数が十分小さい時に高い適合率を維持できることから，本手法は機能フェイズに対応するフェイズ検出に適していると考えられる．

加えて，出力されるフェイズとシナリオ上の機能との対応関係が，シナリオの実行ステップ順や実行時環境，システムの詳細な内部実装の影響を受けにくく安定しているという特徴があり，これは本手法の特に有用な点である．

しかし，出力フェイズ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，機能フェイズの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，本手法で出力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も判明した．このような差異がなぜ発生するのか，特に人間がどのような単位でフェイズを認識するのかを調査していく必要がある．そのほか，今後の課題として，提案手法で分割したフェイズに対し，フェイズごとの特徴を抽出し，また適切な識別名を与えるなどして，それぞれのフェイズが対応する機能とのマッピングを行う必要がある．

謝辞

本研究の全課程を通して，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致します.

本研究を通して，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 松下誠准教授に深く感謝致します.

本研究を通して，逐次適切な御指導および御助言を頂きました 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.

本研究を通して，逐次適切な御指導および御助言を頂きました故 谷口考治氏に深く感謝いたします.

本研究に対して，実験にご協力いただき，また開発現場の視点から貴重なコメントを頂きました株式会社日立システムアンドサービス英繁雄氏，前田憲一氏に深く感謝いたします.

最後に，その他様々な御指導，御助言を頂きました 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 ソフトウェア工学講座 井上研究室の皆様に深く感謝いたします.

参考文献

[1] Cornelissen, B., van Deursen, A., Moonen, L. and Zaidman, A.: Visualizing Test-suites to Aid in Software Understanding. Proceedings of the European Conference on Software Maintenance and Reengineering, pp.213-222, 2007.

[2] Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code. IEEE Transactions on Software Engineering, 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, Proceedings of IEEE International Conference on Program Comprehension, pp.181-190, 2006.

[4] Kollmann, R. and Gogolla, M.: Capturing Dynamic Program Behavior with UML Collaboration Diagrams. Proceedings of the European Conference on Software Maintenance and Reengineering, pp.58-67, 2001

[5] Koschke, R. and Quante, J.: On Dynamic Feature Location. Proceedings of the International Conference on Automated Software Engineering, pp.86-95, 2005.

[6] LaToza, T. D., Venolia, G. and DeLine, R.: Maintaining Mental Models: A Study of Developer Work Habits. Proceedings of the International Conference on Software Engineering, pp.492-501, Shanghai, China, 2006.

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

[8] 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.

[9] Nagpurkar, P., Hind, M., Krintz, C., Sweeney, P. F. and Rajan, V. T.: Online Phase Detection Algorithms. proceedings of Code Generation and Optimization, pp.111-123, 2006.

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

[11] 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.

[12] 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.

[13] Reiss, S. P. and Renieris, M.: Encoding Program Executions. Proceedings of the International Conference on Software Engineering, pp.221-230, 2001.

[14] Reiss, S. P.: Dynamic Detection and Visualization of Software Phases. Proceedings of the third international workshop on Dynamic analysis, pp.50-55, 2005.

[15] Richner, T. and Ducasse, S.: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. Proceedings of the International Conference on Software Maintenance, pp.34-43, 2002.

[16] Rountev, A., Volgin, O. and Reddoch, M.: Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams. Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp.96-102, 2005.

[17] Rountev, A. and Connell, B. H.: Object Naming Analysis for Reverse-Engineered Sequence Diagrams. Proceedings of the International Conference on Software Engineering, pp. 254-261, 2005.

[18] Salah, M. and Mancoridis, S.: A Hierarchy of Dynamic Software Views. From Object-Interactions to Feature-Interactions. Proceedings of the International Conference on Software Maintenance, pp.72-81, 2004.

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

[20] 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.

[21] Tonella, P. and Potrich, A.: Reverse Engineering of the Interaction Diagrams from C++ Code. Proceedings of the International Conference on Software Maintenance, pp.159-168, 2003.

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

[23] Wang, Tao. and Roychoudhury, Abhik.: Hierarchical Dynamic Slicing. Proceedings of the International Symposium on Software Testing and Analysis,] pp.228-238, 2007.

[24] Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering, Vol.18, No.12, pp.1038-1044, 1992.