情報処理学会論文誌　Vol. 51　No. 12　2273–2286 (Dec. 2010)

協調動作するオブジェクト群の変化に基づく実行履歴の自動分割

渡邊　結[†1]　石尾　隆[†1]　井上克郎[†1]

オブジェクト指向プログラムの動作を理解するには，実行時のオブジェクトの振舞いを可視化するアプローチが有効である．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手法やツールは多く提案されているが，巨大な実行履歴から生成された図は読解が困難である．そこで本研究では，1 つの実行履歴を複数のフェイズに自動的に分割する手法を提案する．オブジェクト指向プログラムが 1 つの機能を実行するために多数の一時オブジェクトを生成し，それらの大半を機能の実行完了時に破棄するという性質に基づき，LRU キャッシュを用いて動作するオブジェクト群を観測することでそのキャッシュの更新頻度からフェイズの遷移を検出する．提案手法を 2 つの企業アプリケーションに適用した結果，開発者が 1 つの機能と認識するフェイズを自動的に検出できることを示した．

Automatic Phase Detection for Execution Trace Based on Collaboration of Objects

Yui Watanabe,[†1] Takashi Ishio[†1] and Katuro Inoue[†1]

Visualizing collaborations of objects is important for developers understanding and debugging an object-oriented program. Many techniques and tools are proposed to visualize dynamic collaborations involved in an execution trace of a system, however, such execution trace may be too large to be transformed into a single diagram. In this paper, we propose a novel approach to efficiently detecting phases, or high-level behavioral units described in a use-case scenario. Our idea is based on the nature of object-oriented programming; a phase starts with preparing objects for the phase and ends with destroying temporary objects. Our technique uses an LRU cache for observing a working set of objects, and interprets a sharp rise in the frequency of the cache update as a phase transition. We have applied our approach to two industrial applications and found that our approach is promising to visualize a phase corresponding to a feature as a sequence diagram.

1. まえがき

オブジェクト指向プログラムの動作を理解するには，実行時のオブジェクトの振舞いを実行履歴として記録し，開発者が読み解きやすい形式で可視化するアプローチが有効である[1]．実行時のオブジェクトの振舞いを記録した実行履歴は，実行されたイベントが時系列順にならんだイベント列である．オブジェクト指向プログラムは要求をオブジェクト間のメッセージ通信によって実現するため，実行履歴上のイベントは実行されたオブジェクト間のメッセージ通信イベントであり，実行時刻やメッセージを送信したオブジェクトと受信したオブジェクト，メッセージの内容などの情報を保持している．1 つの実行履歴は通常，複数の機能の実行を連続して記録したものであり，開発者がプログラムの特定の機能の動作を調査する場合，着目する機能を実行に対応するイベント列を実行履歴上の膨大な数のイベントの中から特定しなければならない．従来，このような機能の特定は手作業で行われており，開発者が実行履歴の概要を理解するための手法として，パターン検出を用いて実行履歴の内容を圧縮する手法[6),13] や，可視化するイベントあるいはオブジェクトを選択するクエリを記述する手法[4),20]，縮小された全体図から一部を選択して拡大表示する手法[3),12] などが提案されている．しかし，実行履歴上での機能の実行範囲を特定するには，プログラム全体の構造を理解した上で実行履歴の詳細を読解していく必要があり，これは分析対象に対する知識が乏しい開発者にとって困難な作業である．

これに対して，本研究では 1 つの実行履歴を複数のフェイズに自動的に分割する手法を提案する．**フェイズ**とは，実行履歴上から切り出された連続するイベント列のうち，「入出力処理」や「データベースアクセス」など，開発者にとって意味のある処理に対応するものを指す[14)]．提案手法を用いることで，開発者は着目する機能の実行に対応するフェイズを実行履歴から取り出し，その機能の動作のみを詳細に分析することができる．提案手法で検出するフェイズに対応する機能の粒度は，システムに対する要求を実現する “機能（Feature）” [5),17)] と，それらの “機能（Feature）” の詳細な実行シナリオ上の各ステップに該当する “タスク” の 2 段階とした．

提案手法は，実行履歴上で動作するオブジェクト群を Least Recently Used（LRU）キャッシュを用いて観測する．オブジェクト指向プログラムは，1 つの機能を実行する際に多数の

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

© 2010 Information Processing Society of Japan

中間データ用のオブジェクトを生成し，その機能の実行が終了した時点でそれらのオブジェクトの大半を破棄するという性質[10) を持っている．そのため，メソッド呼び出しイベントに関わったオブジェクトを LRU キャッシュに登録していくことで，キャッシュの更新の頻度から，あるフェイズが終了し次のフェイズが開始したことを検知することができる．

このアルゴリズムは計算コストが低く，大量のイベントとオブジェクトを含む実行履歴を実用的な時間で処理することができるため，実行履歴の解析に適している．一例をあげると，提案手法のプロトタイプの実装を 100 万イベント超の実行履歴に対して適用した場合でも，計算時間は 30 秒程度である．そこで，提案手法はこの計算量の低さを利用し，利用者が実行履歴と求めるフェイズの粒度に対応する出力フェイズ数を入力することで，複数のパラメータセットで本手法を適用した結果から入力された出力フェイズ数に沿う結果を抽出する．

本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．その結果，2 つのパラメータの変化によって出力フェイズ数を制御できることを確認した．また，10 個のフェイズを検出した場合には平均 8 個のフェイズが正しく機能の実行に対応しており，それらは実行履歴中に含まれる “機能（Feature）” に対応するフェイズの 93%，“機能（Feature）” を構成する “タスク” に対応するフェイズの 48%を検出していた．さらに，実装のわずかな差異に影響されず，同一の条件下である履歴から検出されたフェイズに対応する機能が他の履歴からも同様に検出されることを確認した．

以下 2 章で本研究の背景としてプログラムの振舞いの可視化とフェイズの識別について，3 章で実行履歴から自動でフェイズを検出する手法について，4 章で適用実験について述べ，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 オブジェクト指向プログラムの振舞いの可視化

一般に，動的束縛など実行時に決定される要素が多いという特徴がオブジェクト指向言語で記述されたシステムの保守を困難にするといわれている[24)]．開発者がオブジェクト指向言語で記述されたシステムに対する理解やデバッグを行う際には，実行時のオブジェクトの振舞いを実行履歴として記録し，オブジェクト間のメッセージ通信によるオブジェクトの協調動作を開発者が読み解きやすい形式で可視化するアプローチが有効である．オブジェクトの協調動作はクラスよりも大きなプログラム理解の単位であり[15)]，その動作理解やリバースエンジニアリングなどはクラス構造を理解するよりも困難な作業である[1)]．そこで，開発者がオブジェクト指向プログラムの動的な振舞いを理解するため，実行時のオブジェクトの協調動作を可視化する手法やツールが多く提案されている[1),4),6),8),12),20)]．

1 つの実行履歴には実行された複数の機能の実行が連続して記録されているため，ある特定の機能のデバッグやプログラム理解を目的として実行履歴を可視化し分析する場合，開発者は実行履歴中から着目する機能の実行に対応するイベント列を特定しなければならない．そのため，(1) まず開発者は実行履歴全体の概要を調査し，着目する機能が実行されている領域のおおよその位置を推定する．(2) 次に該当領域を詳細に読解し，着目する機能の実行を構成するイベントや，そのようなイベントで動作するオブジェクトを発見する．(3) そのうえで，発見したイベントやオブジェクトを起点に実行履歴を時系列に沿って精読し，機能の実行の開始地点と終了地点を識別することで，着目する機能が実行されている区間を特定する．

この着目する機能の実行区間を特定する作業は，通常開発者が手作業で可視化された実行履歴を読解して行われているが，この際に問題となるのが，実行履歴を構成する膨大な量のイベントとオブジェクトである．(1) で実行履歴の概要を知るためには履歴全体を調査する必要があるが，実行履歴中に含まれるすべての要素をそのまま可視化すると，開発者の読解に耐えない巨大な図となってしまう．また，作業 (2)，(3) で着目する機能の実行区間を特定するためには，システムと着目する機能に関する知識，たとえば実行シナリオ上で示されるシステムの各機能の実行順序や，さらに詳細な設計仕様書で定義される特定の機能の実現にのみ関わるクラスやメソッド，イベント列などを手がかりにする必要があるため，これから対象プログラムやその機能を理解しようとする開発者にとっては困難な作業である．特に作業 (2)，(3) では結果的に着目する機能の実行区間より広い範囲を精読する結果となる場合があり，開発者は解析したい区間を何度も読み直すことになる．加えて，オブジェクト指向プログラムはたとえ同じ機能を実行した場合でも，その実行に対応するフェイズ内に登場するオブジェクトやイベント列は一致しない．このため，たとえばテスト工程である機能のテストのために同じ実行シナリオを同じプログラムの複数のバージョン・複数の環境で実行した場合など，特定の機能に対する複数の異なる実行上の差異を検証する際は，着目する機能の実行区間の特定作業をすべての実行履歴に対して個別に行う必要が生じる．以上の理由により，実行履歴上から着目する機能の実行区間を特定する作業は，対象システムと機能の実装の詳細に関する知識を持つ開発者にとってもコストのかかる工程であるといえる．

この作業を補助するためのアプローチはすでに多く提案されており，それらは以下の 3 種類に分類できる．

© 2010 Information Processing Society of Japan

詳細部分を省略・略記する手法　Hamou-Lhadj ら[6] の手法では，一般に重要でないと考えられるユーティリティクラスのインスタンスに対するものなど，実装の詳細にあたる可能性の高いメソッドコールをフィルタリングする．Reiss ら[13] の手法では，パターン検出を用いて履歴から冗長な箇所を検出し，可視化される図を圧縮する．また，いくつかの実行履歴可視化ツールは，ループや再帰処理などによるメソッドコールの繰返し部分を検出し，略式表示する機能を備えている[1),8),12]．

全体の概要図を表示する手法　Pauw ら[12] は，可視化された図を拡大・縮小する機能を提案し，Cornelissen ら[3] は Circular Bundle View という形式で実行履歴の概要を可視化する手法を提案した．これらを用いることで，開発者は実行履歴をトップダウン方式で探索していくことができる．

開発者の着目するイベントのみを可視化する手法　DRT（design recovery tool）[2] は，解析対象プログラムの GUI 操作から，関連するメソッドコールを自動的に検出する機能を備えている．Shimba[20] と JIVE[4] は，開発者が記述するクエリに該当するイベントだけを可視化する．Briand ら[1] のツールは，分散システム上での遠隔メソッドコール（RMI）だけを可視化する．

上記の手法はいずれも実行履歴を可視化した図の可読性を向上させ，作業 (1) の調査コストを削減することができる．しかし，作業 (2)，(3) は依然として開発者が手作業で行わなければならない．

そこで，本研究では実行履歴から各機能の実行区間（**フェイズ**）を自動で検出する手法を提案する．これは (3) 機能の実行開始・終了地点を特定する作業を自動化するアプローチである．1 つの実行履歴中に含まれる複数の機能の実行区間の開始点を自動検出することで，巨大な実行履歴をフェイズの列に分割する．その結果開発者は，検出されたフェイズから任意のフェイズを選択することで着目する機能の実行区間を特定することが可能となる．このとき開発者はシステムの着目する機能に関する詳細な知識を持っているとは限らないため，実行された機能の数や並び，着目する機能に関連しそうな識別子など，開発者が容易に得ることができる知識のみを用いてフェイズ検出と選択を行うものとして提案手法を設計する．これにより，本手法は着目する機能の実行区間を特定する (1)，(2)，(3) すべての作業に必要な知識・工数コストを削減する．

本研究で提案するフェイズ検出手法は，実行履歴中から開発者が着目する機能を実行した部分のみを可視化・分析することを可能にするものであり，分類上は上述 3 番目の分類「開発者の着目するイベントのみを可視化する手法」に該当するが，実行履歴から各機能の実行に対応する区間を自動で検出できる点で他の手法と異なる．そのため，本手法と他の実行履歴可視化手法・可読性向上手法とを組み合わせて用いることができる．実際に，本研究で提案するフェイズ検出手法は，我々が開発した実行履歴可視化ツール Amida[8] に組み込む形で実装しており，実行履歴全体ではなく検出したフェイズだけを可視化することにより図中の要素を削減し，可視化された図やその縮小図から読解したい部分を特定することを容易にしている．このほかにも，本手法によって検出したフェイズを比較することで各機能に固有の処理の把握を容易にしたり，実行履歴をフェイズの列として抽象化することで大量の実行履歴の管理・検索を容易にしたりすることができると考えられる．

2.2　フェイズの識別

フェイズとは，実行履歴から切り出された連続するイベント列のうち，開発者にとって意味のある処理の実行に対応するものを指す[14]．Salah らは，1 つの機能は連続したイベント列であると述べている[17]．

これらの定義に基づき，本研究ではフェイズに対応する「開発者にとって意味のある処理」をシステムの “機能（Feature）” の実行と各 “機能（Feature）” を構成する “タスク” の実行の 2 段階で取り扱い，それぞれの実行に対応するイベント列を**機能フェイズ（Feature-level phase）**，および**サブフェイズ**として検出する．あるシステムの提供する「機能」が指すものは，要求仕様レベルの機能からメソッドレベルの機能まで様々である．そのうち，デバッグやプログラム理解を目的として実行履歴を可視化し分析する際に「開発者が着目する機能」とは，多くの場合システムに対する要求を実現する “機能（Feature）” [5),17] である．これらの “機能（Feature）” は，対象ソフトウェアの利用者用ドキュメント中の記述や設計仕様書におけるユースケースなどで示される，システムの要求仕様や上位設計に対応する．ただし，対象プログラムや実行された “機能（Feature）” に対してある程度知識を持った開発者が着目する機能は，より詳細な粒度の機能を指す場合がある．そこで，システムの “機能（Feature）” の子要素として “タスク” を定義する．“タスク” は，ある 1 つの “機能（Feature）” をさらに詳細な実行ステップとして分解したときの，1 つの実行ステップに対応する単位である．一例をあげると，4 章の適用実験で使用した実行シナリオ T-1 は**表 1** に示す 5 つのステップを順に実行するものであり，この 5 つのステップはそれぞれ対象プログラムの持つ 5 つの “機能（Feature）” の実行によって実現される．また，“機能（Feature）” の 1 つである「Login」の詳細な実行シナリオは，表 1 に示す異なる 4 つの “タスク” を順に実行することで実現される．表 1 により，この実行シナリオ T-1 はシステムの持つ 5 つの “機能（Feature）”，もしくは 14 個の “タスク” によって実現されることが分かる．この

© 2010 Information Processing Society of Japan

表 1　実行シナリオ T-1 の内容と各ステップで実行される機能（“機能（Feature)”，“タスク”）
Table 1　Features and Tasks executed in scenario T-1.

ステップ	“機能（Feature)”	“タスク”
1. ログインする	Login	Show login form Login Get pre-user settings Show entrance page
2. 登録情報の一覧を表示する	Listing tools	Get management information Get pre-user items Get list of items Show list of items
3. 選択項目の内容を表示する	Maintenance view of a tool	Get an item ID Get a detail of the item show the item information
4. 選択項目の内容を変更する	Updating the tool information	Get an item ID Update the item information Get a detail of the item Show the item information
5. ログアウトする	Logout	Logout Show the login form shutdown the system

ように，プログラムの実行シナリオは複数の “機能（Feature)” の実行によって実現され，また “機能（Feature)” は 1 つ以上の “タスク” で構成される．ただし例のように，共通の “タスク” が異なる複数の “機能（Feature)” の構成要素となる場合もある．ソフトウェアの設計段階で，開発者が必ず “機能（Feature)” を “タスク” に分解して定義することを保証することはできないが，少なくとも本研究の適用実験で使用したシステムを制作した 2 つの企業においては，各 “機能（Feature)” に対してその詳細な実行シナリオを複数のステップに分解して記述する方針を採用していることから，現実的な仮定であると考えている．

実行履歴をフェイズごとに分割する場合，着目する機能の粒度によって分割結果が異なる．本稿では，システムへの要求を実現する 1 つの “機能（Feature)” の実行に対応するフェイズを**機能フェイズ（Feature-level phase）**と呼び，“タスク” の実行に対応するフェイズを**サブフェイズ（Minor phase）**と呼ぶ．**機能フェイズ**はシステムに対する要求を実現する 1 つの “機能（Feature)” が実行されたとき，その開始から終了までに対応する実行履歴上の区間である．1 つの実行履歴は実行シナリオ上の各ステップに対応する機能フェイズがシナリオ上で示される順に並んだ列として表すことができる．**サブフェイズ**は 1 つの “機能（Feature)” を実現するためのユースケースシナリオを構成する “タスク” の実行の開始から終了までの区間である．実行履歴上の各機能フェイズは 1 つ以上のサブフェイズの列で構成される．先ほど例にあげた実行シナリオ T-1 を実行し記録した実行履歴から，各メソッド呼び出しイベントで動作したオブジェクトをプロットしたグラフを**図 1** に示す．X 軸は実行履歴上の時間軸を表し，イベントの発生順序を示す．Y 軸は各イベントにおける呼び出し先オブジェクトおよび呼び出し元オブジェクトの ID を表す．この実行履歴は実行されたオブジェクト間のメッセージ通信イベントで構成されるイベント列であるが，表 1 に示す 5 つの “機能（Feature)” を順番に 1 度ずつ実行するため，この実行履歴は対応する 5 つの機能フェイズに分割することができる．図 1 ではこれら 5 つの機能フェイズごとに背景色を区切っている．さらにこのシナリオは合計 14 個の “タスク” で構成されているが，そのうち 4 つの “タスク” が異なる機能フェイズでそれぞれ実行されているため，この実行履歴は 18 個のサブフェイズの列となる．

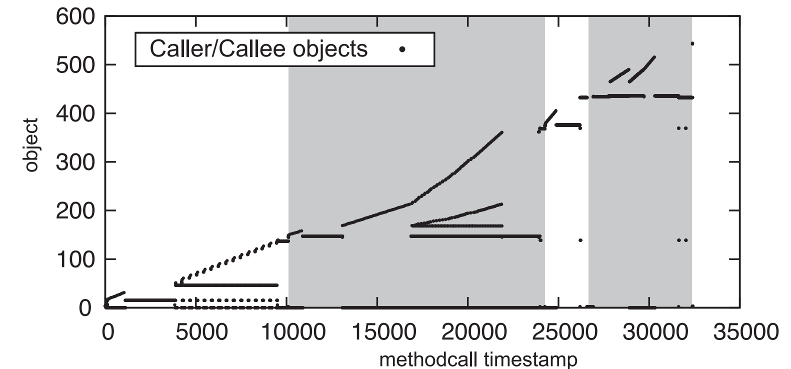


図 1　5 つの機能フェイズを含む実行履歴を構成するメソッドコールと各呼び出し元/呼び出し先オブジェクト
Fig. 1　Caller/Callee Objects in a use-case scenario of an industrial system. The execution trace comprises five feature-level phases: login, three steps to update a database record and logout.

サブフェイズもまたより小さなフェイズの列で構成されると見なすことはできるが，本研究では開発者にとって意味のある処理に対応するフェイズの検出を目的とするため，機能フェイズとそれを構成するサブフェイズの 2 段階の粒度のフェイズのみを対象とする．フェイズにはこれ以外の定義も存在しており，たとえば Wang らは構文構造に基づくフェイズを定義し，それ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．Wang らが提案した構文構造に基づくフェイズは階層関係を持ち，開発者が実行履歴から障害の詳細を

© 2010 Information Processing Society of Japan

追う場合などに役立つが，ユースケースに示される機能との対応関係は保証されない．また，プログラムの性能計測や最適化の分野においては，フェイズとは実行を一定の間隔（実行時間上で 10 ミリ秒[14] ごとなど）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[11] も存在するが，いずれもソフトウェアの機能に対応するものではなく，本手法で検出するフェイズとは異なる．

3. フェイズの自動検出

本研究の目的は，1 つの実行履歴から実行された “機能（Feature)” あるいは “タスク” ごとの実行区間を特定することである．提案手法は，入力された実行履歴から機能フェイズまたはサブフェイズの開始点となるイベントを抽出することで，フェイズの自動検出を行う．

3.1 フェイズ検出アプローチ

提案手法では，協調動作するオブジェクト群の切り替わりによってフェイズの移行を検出し，コールスタックの深さによってフェイズの開始点を 1 点に決定する．このアプローチは，オブジェクト指向プログラムにおける以下の 2 つの既知の性質を利用したものである．

性質 1 オブジェクト指向プログラムでは，ある 1 つの意味のある処理（“機能（Feature)” や “タスク”）を実行する際に多数の中間データ用のオブジェクトが生成され，またその処理の実行が終了した時点でそれらオブジェクトの大半が破棄される[10),22]．

性質 2 オブジェクト指向プログラムを実行したとき，機能の実行の開始はメソッド呼び出しイベントに対応する．

性質 1 より，あるフェイズで動作するオブジェクトの多くはフェイズ内で新たに生成されたオブジェクトであり，その大半は機能の実行が終了すれば破棄されるため，複数のフェイズにまたがって登場するオブジェクトは少数であり，各機能はそれぞれのフェイズ中に生成され破棄される固有のオブジェクト群が協調動作することによって実現されていると考えられる．

図 1 は，本研究の適用実験に用いたシナリオ T-1 を実行した際のメソッドコールイベントの履歴をプロットしたものである．X 軸は実行履歴上の時間軸を表し，メソッドコールイベントの発生順序を示す．Y 軸は各メソッドコールイベントにおける呼び出し先オブジェクトおよび呼び出し元オブジェクトの ID を表す．この実行履歴は「ログイン」「DB 内容一覧」「選択項目の編集」「編集のコミット」「ログアウト」という 5 つの “機能（Feature)” の実行を記録したものであり，図 1 ではこれら 5 つの “機能（Feature)” に対応する機能フェイズごとに背景色を区切っている．この実行履歴から，ある “機能（Feature)” や “タスク” の実行に対応するフェイズはそのフェイズ固有のオブジェクト群によるイベントで構成されており，フェイズが移行すれば動作するオブジェクト群の大半が新しいフェイズ固有のオブジェクトへ入れ替わっていることが読み取れる．

このように，機能フェイズの移行は協調動作するオブジェクト群の局所的かつ大規模な入れ替わりをともなうことから，実行履歴上の時間経過にともなうオブジェクト群の変化を監視することでフェイズの移行を自動で検知することが可能であると考える．そこで提案手法は **Least-Recently-Used（LRU）**キャッシュを用いて動作するオブジェクト群の変化を監視する．LRU キャッシュアルゴリズムは他のアルゴリズムと比較して，実行履歴上の小さな 1 区間で局所的に動作するオブジェクト群を捕捉する処理に適している[18]．1 つの機能フェイズの実行中は，動作するオブジェクト群がそのフェイズ固有のオブジェクトで安定するために，LRU キャッシュの更新の頻度が下がる．実行が次の機能フェイズへ移行すると，協調動作するオブジェクト群が新しいオブジェクトへいっせいに入れ替わるため，LRU キャッシュが頻繁に更新される．そこで提案手法では，LRU キャッシュの更新頻度の上昇によって新しいフェイズの開始を自動的に検知する．

LRU キャッシュを用いて機能フェイズの移行を検知した後，そこから新しい機能フェイズが開始した点を実行履歴上の 1 点で識別する必要がある．性質 2 より，フェイズの開始点はメソッドコールイベントである．加えて，性質 1 よりあるフェイズが終了する際にはフェイズ固有のオブジェクトの大半が破棄されるため，フェイズ固有オブジェクトからのメソッドリターンイベントが多数出現し，続いて新しいフェイズが開始すると新しいフェイズ固有のオブジェクト群に対するメソッドコールイベントが多数出現すると考えられる．したがって，フェイズの開始点となるメソッドコールイベントではコールスタックの深さが局所的最小値をとると考えられる．ただし，実行履歴上にはコールスタックの深さが局所的最小となる時刻が数多く存在し，それらの多くは機能フェイズやサブフェイズの開始点とは無関係である．

そこで，提案手法ではオブジェクト群の変化を捕捉する LRU キャッシュの更新頻度の上昇によりフェイズの移行を検知した後，その時刻の直前の一定範囲のイベントのうちコールスタックの深さが局所的最小となるメソッドコールイベントを新たなフェイズの開始点として検出する．1 つの機能フェイズの開始に対応する LRU キャッシュの更新頻度の上昇が複数イベントで検知される可能性があるが，それぞれのイベントから直前のコールスタックの深さが局所的最小となるメソッドコールイベントを探索することで，フェイズの開始点を履歴上の 1 点に定めることができる．

© 2010 Information Processing Society of Japan

```
procedure  DetectPhases (
              in [e_1, e_2, ··· e_last]: list of methodcall event;
              in c, w, m: integer;
              in threshold: double;
              out P : set_of_integer ) ;

(01)  LRUCache C = new LRU_cache(c)
(02)  integer[] update = []
(03)  double[] frequency = []
(04)  set of integer P = φ

(05)  for integer t in [1 . . . last]

(06)     b1 = C.update(e_t.caller, t)
(07)     b2 = C.update(e_t.callee, t)
(08)     if b1 ∨ b2 then update[t] = 1 else  update[t] = 0

(09)     double flags = 0
(10)     for integer k in [max(t − w + 1, 1) . . . t]
(11)        flags = flags + update[k]
(12)     end for
(13)     frequency[t] = flags/w

(14)     if frequency[t] ≥ threshold then
(15)        integer min = max(t − m + 1, 1)
(16)        for integer k in [max(t − m + 1, 1) . . . t]
(17)           if e_min.callstack ≥ e_x.callstack then min = k
(18)        end for
(19)        P = P.add(min)
(20)     end if

(21)  end for
```

図 2　フェイズ検出アルゴリズム
Fig. 2　Phase detection procedure.

3.2　フェイズ検出アルゴリズム

提案手法のアルゴリズムを**図 2** の手続き DetectPhases に示す．提案手法は，入力として実行履歴から抽出されたメソッドコールイベント列 $E = [e_1, e_2, \cdots e_{last}]$ と，4 つのパラメータ c，w，m，$threshold$ を受け付ける．

実行履歴中の各メソッドコールイベント e_k は，イベント時刻 k，呼び出し先オブジェクト $e_k.callee$，呼び出し元オブジェクト $e_k.caller$，コールスタックの深さ $e_k.callstack$ の 4 つの情報を持つ．**イベント時刻**（k）はメソッドコールイベントが発生した順序を表す整数値である．1 つの履歴中に存在するすべてのイベントは，異なるスレッド上で発生した場合においても，直列化を行いそれぞれ一意なタイムスタンプを持つ．**呼び出し先オブジェクト**（$e_k.callee$）はメソッドコールイベントにおいて呼び出された側のオブジェクトを一意に識別する ID を表す．**呼び出し元オブジェクト**（$e_k.caller$）はメソッドコールイベントにおいて，呼び出した側のオブジェクトを一意に識別する ID を表す．**コールスタックの深さ**（$e_k.callstack$）はそのメソッドコールイベントが発生した時点でのコールスタックの深さを表す．

提案アルゴリズムは，上記の実行履歴情報のほかに，以下の 4 つのパラメータを持つ．

キャッシュサイズ c　図 2 の手続き DetectPhases において (01) 行で生成される固定長の LRU キャッシュ C のサイズを表す整数値である．最小値は 1 であり，最大値は実行履歴に登場するオブジェクトの数に等しい．
キャッシュサイズ c は本手法が 1 つのフェイズを構成すると見なすオブジェクト群の要素数を表すため，キャッシュサイズ c を大きくすることでより規模の大きいフェイズ（機能フェイズ）が検出され，小さくすることでより規模の小さいフェイズ（サブフェイズ）が検出されるようになると考えられる．したがって，キャッシュサイズ c は検出されるフェイズの粒度に影響を及ぼす．

ウィンドウサイズ w　各イベント時刻における LRU キャッシュ C の更新頻度 $frequency$ を計算する範囲を表すイベント数である（図 2 (10)，(13)）．最小値は 1 であり，最大値は実行履歴に含まれるメソッドコールイベントの数（図 2 における $last$）に等しい．
ウィンドウサイズ w の値が大きいほど各イベントにおける LRU キャッシュの更新頻度の値の変化はなだらかになるため，フェイズ移行にともなって更新頻度が上昇する時刻がフェイズ開始点から遠ざかり，またその値も小さくなる．したがって，ウィンドウサイズ w を小さくするほど検出されるフェイズの数が増加し，大きくするほど検出されるフェイズの数が減少すると考えられる．

閾値 $threshold$　各イベント時刻における LRU キャッシュ C の更新頻度 $frequency$ の値がフェイズ移行を表す下限値として用いる閾値である（図 2 (14)）．最小値は 0 であり，最大値は 1 である．
閾値 $threshold$ はウィンドウサイズ w とともに，あるイベントの直前に連続するメソッドコールイベント w 個のうちどれだけの割合のイベントで LRU キャッシュの更新が生じた場合に「フェイズの移行が生じた」と見なすかを決定する値である．そのため，ウィンドウサイズ w と同様に閾値 $threshold$ の値を小さくすることで検出されるフェイズの数が増加すると考えられるが，ウィンドウサイズ w と異なり各イベント時刻における LRU キャッシュ C の更新頻度 $frequency$ の値そのものには影響を与えない．一方で，更新頻度 $frequency$ の値が閾値 $threshold$ を超えたイベントを起点としてフェイ

© 2010 Information Processing Society of Japan

ズの開始点を探索するため，フェイズ開始点を探索する範囲の決定には影響を与える．

スコープサイズ m　フェイズ移行が検知された際，新しいフェイズの開始点を探索する範囲を指定するイベント数である（図 2 (15)，(16)）．最小値は 1 であり，最大値は実行履歴に含まれるメソッドコールイベントの数（図 2 における $last$）に等しい．

閾値とともに，フェイズ開始点となるメソッドコールイベントを探索する範囲に影響を与える．フェイズ開始点となりうるコールスタックの深さが局所的最小となる点は実行履歴上で固定であるが，LRU キャッシュ C の更新頻度 $frequency$ の上昇が検知されるイベント時刻と，対応するフェイズの開始点の距離は他の 3 つのパラメータによって変化するため，スコープサイズ m の値を適切に設定しなかった場合，正しいフェイズ開始点を検出できなくなる可能性がある．

提案手法のアルゴリズムを図 2 の手続き DetectPhases に示す．提案手法は，メソッドコールイベント列 $E = [e_1, e_2, \cdots e_{last}]$ に対し，時系列順に以下の処理を施すことによってフェイズを検出する．

1．動作するオブジェクト群の観測（図 2 (05)–(08)）．まず，実行履歴中のメソッドコールイベント e_t の呼び出し元オブジェクトと呼び出し先オブジェクトを，時系列順にキャッシュ C へ追加し，各時刻におけるキャッシュの更新の有無を判定する．

図 2 (06)–(07) の $LRUCache.update(objectID, timestamp)$ では，第 1 引数として与えられたオブジェクトを Least Recently Used アルゴリズムに基づいてキャッシュに追加し，キャッシュの更新の有無を論理型で返す．

第 1 引数として与えられたオブジェクトがキャッシュ内に存在しない場合は，このオブジェクトを最新の要素としてキャッシュへ追加し，これによりキャッシュを更新したと見なして真を返す．新たなオブジェクトを追加したことによりキャッシュ内のオブジェクト数がキャッシュサイズ c を超えてしまう場合は，キャッシュ内で最も古い要素，すなわち至近の登場時刻が最も早いオブジェクトを破棄する．

一方，第 1 引数として与えられたオブジェクトがすでにキャッシュ内に存在する場合は，キャッシュの更新は行わず，偽を返す．ただしこのとき，キャッシュ内にすでに存在する該当オブジェクトの登場時刻を更新し，キャッシュ内で最新の要素とする．

2．フェイズ移行の検知（図 2 (09)–(13)）．フェイズの移行を表す値として，履歴上の各イベント時刻 t における LRU キャッシュ C の**更新頻度**（$frequency(t)$）を以下に定義する．

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

$updated[k]$ は，イベント時刻 k でキャッシュが更新された場合に 1，更新されなかった場合に 0 の値をとる．$frequency(t, w)$ は過去 w 回のメソッドコールイベント中でキャッシュの更新が生じたイベントの割合を表し，この値がパラメータとして与えられた閾値 $threshold$ より高い場合，フェイズ移行が生じたと見なす．

3．フェイズ開始点の識別（図 2 (14)–(20)）．コールスタックの深さ情報を用いて新しいフェイズの開始点となるイベントを識別する．新しいフェイズの開始点となるイベントは，コールスタックの深さが局所的最小値をとるものであるため，フェイズの移行が検知されたイベントから過去 m 回分のイベント列でコールスタックの深さを比較し，最小値をとるイベントをフェイズの開始点として検出する．最小値をとるイベントが複数存在する場合は，最も新しいイベントを選択する．

すべてのメソッドコールイベントに上記の処理を適用し，検出したフェイズの開始点に該当するイベント時刻の列 $P = [t_1, t_2, \ldots, t_p]$ を出力する．

3.3　フェイズ検出手法の特徴と適用方法

提案するアルゴリズムの計算量は，入力される実行履歴中のメソッドコールイベント数を n としたとき $O(nm)$ である．また，このアルゴリズムが消費するメモリ領域は，LRU キャッシュ領域とウィンドウサイズ分のイベント情報を保持する領域分のみであり，これは実行履歴中のメソッドコールイベント数 n と比較して十分に小さい値である．

提案手法では，パラメータの変化が出力されるフェイズに影響を与えると考えられる．正しい出力結果は利用者が求める粒度と入力する実行履歴に依存して定まるため，適切なパラメータを自動で設定することは困難である．そのため利用者が手動でパラメータを設定する必要があるが，この場合は着目する機能の粒度に対応する出力が得られるパラメータセットを探すために本手法を繰り返し適用することになる．各パラメータの値域は有限であるが，対象ソフトウェアに関する知識のない開発者が出力結果の成否判定を行うためには結果的に実行履歴中から出力されたフェイズの境界領域を精読することとなり，着目する機能の実行区間を特定する作業コストを増加させてしまう．

そこで提案手法の利用は，各パラメータを指定する代わりに求めるフェイズの粒度に対応する出力フェイズ数を入力させ，複数のパラメータセットで本手法を適用した結果から入力された出力フェイズ数に沿う結果を抽出するものとする．このとき，もし各パラメータの変化と出力フェイズの数ならびに粒度に相関関係があるならば，開発者が着目する機能の粒度に対応する出力を提示することができると考えられる．

利用者が求めるフェイズの粒度に対応する出力フェイズ数は，入力する実行履歴の実行シ

© 2010 Information Processing Society of Japan

ナリオから容易に算出可能である．各パラメータの値域は有限であり，本手法の計算量は十分小さいことから，複数のパラメータセットで繰り返し本手法を適用するための計算コストは人間が手動で行う作業コストに対して問題にならない．

提案手法の出力フェイズ数をもとに開発者の求めるフェイズの粒度に対応した出力を得るためには，各パラメータの変化に対する出力フェイズの数と粒度とに相関関係があることが前提である．具体的には，指定された出力フェイズ数が十分少ない（入力する実行履歴に含まれる機能フェイズ数またはサブフェイズ数以内である）場合に，以下 3 つの仮定が成り立つ必要がある．

仮定 1 パラメータの変化と出力フェイズ数の変化に相関関係がある．このとき，1 つの大きいフェイズが複数の小さいフェイズに分割される形で出力フェイズ数が増加し，また複数の小さいフェイズが 1 つの大きいフェイズに統合される形で出力フェイズ数が減少する．

仮定 2 機能フェイズやサブフェイズが，本手法で対象としないより小さい粒度のフェイズや誤検出よりも優先して出力される．

仮定 3 機能フェイズがサブフェイズよりも優先して出力される．

これら 3 つの仮定について，4 章の適用実験で検証する．

なお，提案手法をマルチスレッドを用いたプログラムの実行履歴に適用する場合，各スレッドに存在するコールスタックの扱いが問題となる．これに対しては，各スレッドに本アルゴリズムを個別に適用する方法と，すべてのスレッドが単一のコールスタックを共有していると見なし，共有コールスタックの深さをスレッドごとのコールスタックの深さの和とする方法がある．4 章の適用実験では，後者の方法を採用している．

4. 適用実験

開発者が対象ソフトウェアや実行履歴に関する詳細な知識を用いることなく提案手法を利用するための 3 つの仮定を検証するため，実行履歴を入力として提案手法が自動で検出するフェイズと，開発者が実行履歴を手動で読み解いて決定したフェイズとを比較する実験を行った．

4.1 実験対象

実験対象は下記の 2 つの Java システムから Amida Profiler [21] を用いて取得した複数の実行履歴である．なお，観測対象からは Java SDK ライブラリに含まれるクラスのオブジェクトに関するメソッドコールイベントが除外されている．

表 2 実行履歴を取得したシナリオ（機能フェイズ）
Table 2 Use case senarios (feature-level phases) to record execution traces.

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
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

- **ツール管理システム**は，ある企業が開発し，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，ユーザインタフェースに JSP を用いており，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．実験では，**表 2** に示す 4 つのシナリオをそれぞれ実行して実行履歴 T-1，T-2，T-3，T-4 を取得した．「Login」のように複数のシナリオに含まれる “機能（Feature）” も存在するが，実際のメソッド呼び出し列は実行履歴ごとに異なる．
- **書籍管理システム**は，ある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，同一の設計を異なるグループが実装した 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 2 に示す共通のシナリオを実行し，実行履歴 L-1，L-2，L-3，L-4，L-5 を取得した．シナリオは共通であるがそれぞれプログラムの実装が詳細設計レベルで異なるため，1 つの機能フェイズに対応するサブフェイズがプログラムごとに異なる場合がある．そのため，実行履歴ごとに機能フェイズの数や順序は等しいが，サブフェイズの数や順序，対応する “タスク” が異なる．

取得した各実行履歴を各システムの開発者が手動で解析し，実行したシナリオに対応する各機能フェイズおよびサブフェイズの正確な位置（フェイズの開始点）を決定した．システムの開発者が認識したフェイズ数を**表 3** に示す．**#e** は実行履歴に記録されたイベント数（メソッドコール回数）であり，**#obj** は実行履歴に登場するオブジェクトの総数を表す．**#f** は実行履歴を構成する機能フェイズの数を表し，これは実行したシナリオ上のステップ

© 2010 Information Processing Society of Japan

表 3　各実行履歴の詳細
Table 3　Execution traces and number of phases detected by developer in those traces.

ツール管理システム

ID	#e	#obj	#f	#m
T-1	32,416	546	5	18
T-2	30,494	524	6	19
T-3	26,603	438	4	14
T-4	15,909	237	3	10

図書管理システム

ID	#e	#obj	#f	#m
L-1	3,573	261	15	52
L-2	3,371	272	15	51
L-3	3,797	286	15	51
L-4	3,862	300	15	51
L-5	4,506	341	15	64

（表 2）の個数と一致する．**#m** は実行履歴上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．これは実行された “機能（Feature）” に該当するユースケース文書に書かれた実行ステップ数の合計に相当する．どちらのシステムも「システムはパスワードを検証する」といったように，1 つの “機能（Feature）” を実現するためのシステムの動作をある一定の粒度に分解して記述していたので，その文章数を単純にステップ数とした．

4.2 実験手順

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，キャッシュサイズとウィンドウサイズは，実行履歴上の各時刻における LRU キャッシュの更新頻度 $frequency$ の値に直接影響を与える．更新頻度 $frequency$ が実行履歴上の時刻に対応してどのような波形をとるかによって，検出されるフェイズの数や粒度が決定するため，この 2 つのパラメータが出力に与える影響は大きいといえる．そこで，各システムについてキャッシュサイズとウィンドウサイズの複数の組合せで手法を適用し，パラメータの変化に対する出力の変化を検証する．ツール管理システムの各実行履歴に対しては，c を 10 から 600 まで 10 刻みで，w を 10 から 200 まで 10 刻みで変化させた 1,200 通り，図書管理システムの各実行履歴に対しては，c を 10 から 350 まで 10 刻みで，w を 10 から 300 まで 10 刻みで変化させた 1,050 通りのパラメータ設定で本手法を適用した．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に基づいて定めた．w の値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．

残り 2 つのパラメータ，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．これらの値はコールスタックの深さに基づくフェイズ開始イベントの探索範囲に影響する値であり，実行履歴上の各時刻における LRU キャッシュの更新頻度 $frequency$ の値を変化させるキャッシュサイズとウィンドウサイズに比べ，提案手法の出力に与える影響がごく小さいためである．閾値 $threshold$ は更新頻度 $frequency$ の変化を最大限出力に反映させることができるよう十分小さく，かつ 1 つのフェイズの実行中の更新頻度 $frequency$ の一時的な変化によるノイズを除去するために十分大きい値を設定した．予備実験により，更新頻度 $frequency$ は機能フェイズの移行にともなう協調動作するオブジェクト群の変化によって局所的に上昇し，1 つのフェイズの実行中は固定値 0 をとるわけではなく，0 と 0 に近い一定値の間で安定することが確認されている．この値はウィンドウサイズの値によって変化するが，今回の実験ではウィンドウサイズ w の最小値を 10 としたことから，連続する 10 回のメソッドコールイベントのうち 1 回で LRU キャッシュの更新が行われた場合の 0.1 を基準とすることができる．また，スコープサイズ m が検出結果に与える変化は他のパラメータよりも小さいといえる．これは，実行履歴中のコールスタックの深さが局所的最小値となる点の集合は履歴ごとに固定であり，キャッシュサイズ c とウィンドウサイズ w の値を固定した場合，すなわち実行履歴上の各時刻の更新頻度 $frequency$ が与えられた場合に，スコープサイズ m の値を極端に小さくしない限り（特に，ウィンドウサイズ w と同等かそれ以上である限り）検出されるフェイズ開始点の位置が変化しないためである．実験では，スコープサイズ m の固定値が結果に与える影響をさらに小さくするため，実験で用いるすべての実行履歴のメソッドコールイベント数より十分小さく，かつウィンドウサイズ w より十分大きい値として固定値 $m = 700$ を採用した．

ツール管理システムの 4 つの実行履歴それぞれにつき 1,200 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 1,050 通りのパラメータ設定で，提案手法によるフェイズ検出を計 10,050 回実行した．フェイズ検出アルゴリズムの演算は Xeon 3.0 GHz の CPU を備えた計算機上で行い，合計で 5 分間を要した．

4.3 評価基準

収集した出力結果から，3 つの仮定の評価を以下の基準で行う．

仮定 1　各実行履歴に対してキャッシュサイズとウィンドウサイズのうち 1 つのパラメータの増減に対する出力フェイズ数の変化により評価する．このとき，出力フェイズ数の増減がフェイズの分割・統合の形で行われることを確認する．

仮定 2　各実行履歴の出力結果について，出力フェイズ数が実行履歴中の機能フェイズ数，サブフェイズ数よりも小さいとき，開発者が判断したフェイズとの適合率の出力フェイズ数ごとの平均値により評価する．

仮定 3　各実行履歴の出力結果について，出力フェイズ数が実行履歴中の機能フェイズ数，

© 2010 Information Processing Society of Japan

サブフェイズ数よりも小さいとき，開発者が判断したフェイズとの再現率の出力フェイズ数ごとの平均値により評価する．このとき，開発者が判断した機能フェイズとの再現率が，開発者が判断したサブフェイズに対する再現率よりも高いことを確認する．

評価基準として，以下に示す適合率 $precision$ と再現率 $recall$ 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 X の要素数を $|X|$ と表記している．

提案手法の検出結果では，再現率よりも適合率が重要となる．検出できなかったフェイズ，すなわち，いつくかの機能の実行区間を単一のフェイズとして検出してしまう誤りは，開発者がそのフェイズを読解する際のコストが最小化されないという問題はあるものの，フェイズ単位で実行履歴を解析する段階で吸収される．しかし，開発者が認識する機能の実行区間と無関係な箇所で実行履歴を分断するようなフェイズを検出してしまった場合，開発者が選択したフェイズを詳細に読解したとしても，着目した機能の実行の一部がそのフェイズに含まれず，動作理解に必要な情報を読み取ることができない危険性がある．そのため，提案手法の検出するフェイズの境界は開発者の認識する機能の実行区間の境界と正確に一致していなければならない．

4.4 実 験 結 果

4.4.1 出力フェイズ数の変化

履歴 T-1 に対して本手法を適用した際，キャッシュサイズ c とウィンドウサイズ w の 2 つのパラメータの値の変化に対して，検出されるフェイズの変化を**図 3** に示す．左のグラフは $w = 50$ の場合のキャッシュサイズと検出されたフェイズ開始点との関係を表し，右のグラフは $c = 300$ の場合のウィンドウサイズと検出されたフェイズ開始点との関係を表す．グラフの X 軸は実行履歴上の時刻，Y 軸はパラメータの値を示しており，上部のグラフにおける座標 (x, y) の×印は $c = y$ のときの出力にイベント時刻 x が含まれていたことを示している．

図 3 より，キャッシュサイズの値を小さくするとキャッシュの更新が起こる確率が上がるため本手法で検出されるフェイズも小さくなり，出力フェイズ数が増加することが分かる．同様に，ウィンドウサイズの値が小さいほどフェイズ移行時に新しく登場するオブジェクトが少なくても検知しやすくなるため，出力フェイズ数が増加している．このように，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負の相関関係が

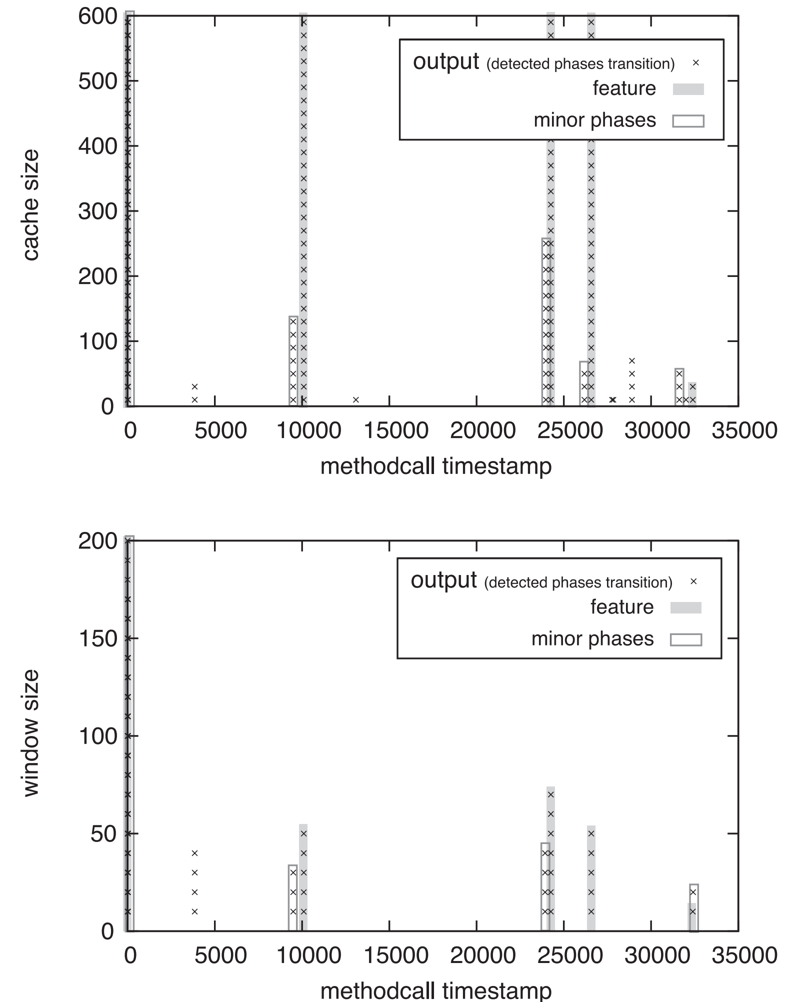


図 3　履歴 T-1 におけるキャッシュサイズおよびウィンドウサイズと出力フェイズ

Fig. 3　Detected phases in the trace T-1. Window size is fixed ($w = 50$) in the upper figure, cache size is fixed ($c = 300$) in the lower figure.

© 2010 Information Processing Society of Japan

ある．

また，図 3 では，キャッシュサイズを $c = 300$ から $c = 200$ へ変化させたとき，検出されたフェイズの数が 1 つ増えているが，これは $c = 300$ のときに検出されたフェイズ開始点は保存されたうえで，そのフェイズのうちの 1 つが 2 分割されたためである．このように，パラメータの変化にともなって出力フェイズ数が増減するとき，個々のフェイズは分割・統合される形で変化している．すべての実行履歴について，キャッシュサイズとウィンドウサイズがある一定値より大きいとき，一方の値を小さくすることで出力フェイズ数が増加し，このときすでに検出されたフェイズ開始点は変化せずに新しいフェイズ開始点が追加される（フェイズが分割される）ことを確認した．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た場合（キャッシュサイズが $c \leq 20$ の場合など）はこの限りではなく，パラメータが大きい場合に検出していたフェイズの開始点を検出しなくなることがある．しかし同時に出力フェイズ数が爆発的に増加するため，機能フェイズ数またはサブフェイズ数をもとに出力フェイズ数を指定した場合に除外される．

4.4.2　適　合　率

図 4 は，履歴 T-1 に対する 1,200 通りのパラメータによる本手法の適用結果の出力フェイズ数ごとの平均適合率・平均再現率を表す．X 軸は出力フェイズ数，Y 軸は適合率と再現率を表す．また，ツール管理システムの 4 つの実行履歴において，出力フェイズ数が 5 個と 10 個であった場合の適合率・再現率の平均値と，書籍管理システムの 5 つの実行履歴において，出力フェイズ数が 10 個，15 個，20 個であった場合の適合率・再現率の平均値を**表 4** に示す．#p は出力フェイズ数，Recall(F) は機能フェイズに対する平均再現率，Recall(A) はサブフェイズを含めた平均再現率，Precision は平均適合率である．

図 4 の Precision(All) は出力フェイズ数に対する平均適合率の変化を表しており，出力フェイズ数が少ないほど高い適合率を示している．実行履歴 T-1 中のサブフェイズ数 18 以下の場合 60%以上，機能フェイズ数 5 以下の場合は 90%以上の高い平均適合率を示した．特に，出力フェイズ数が 1 以上 8 以下となった組合せで適合率が 100%となる出力が多数存在した．

履歴 T-1 を含めすべての実行履歴に対する適用結果で同様の傾向を確認し，特に出力フェイズ数が実行履歴中の機能フェイズ数前後の場合に高い適合率を示した（表 4）．ツール管理システムの場合，各履歴の機能フェイズ数は 3 から 6 個であり，4 つの実行履歴で出力フェイズ数が 5 つの場合の平均適合率は 93%，書籍管理システムの場合はすべての履歴において機能フェイズ数が 15 個であり，本手法の出力フェイズ数が 15 個の場合の平均適合

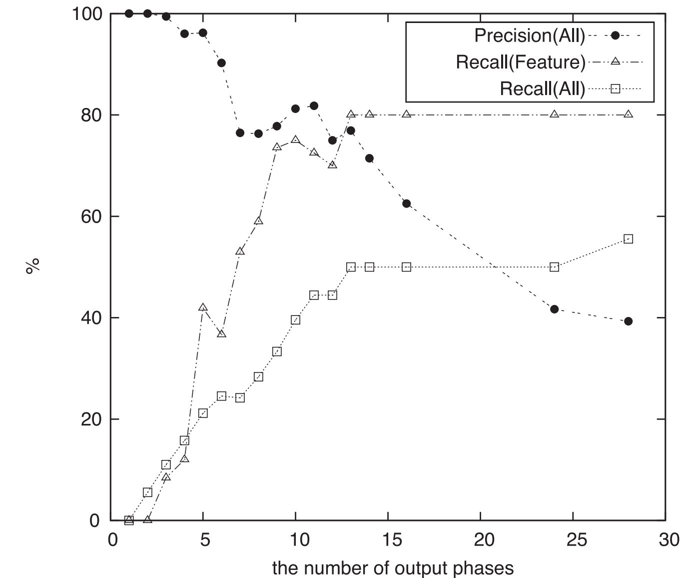


図 4　履歴 T-1 における，出力フェイズ数ごとの平均適合率および平均再現率

Fig. 4　Average recall and precision for various parameter configuration that result in the same number of phases in the trace T-1.

表 4　すべてのパラメータの組合せに対する，出力フェイズ数ごとの平均適合率・平均再現率

Table 4　Average recall and precision for various parameter configuration that detects the same number of phases.

ツール管理システム

#p	Recall(F)	Recall(A)	Precision
5	0.56	0.39	0.93
10	0.90	0.48	0.80

図書管理システム

#p	Recall(F)	Recall(A)	Precision
10	0.24	0.20	0.99
15	0.53	0.29	0.98
20	0.45	0.38	0.96

率は 98%である．

4.4.3　再　現　率

図 4 の Recall(Feature) と Recall(All) は，それぞれ出力フェイズ数ごとの平均再現率の変化を示す．Recall(Feature) は履歴 T-1 に含まれる 5 つの機能フェイズに対する再現率であり，Recall(All) はサブフェイズを含むすべてのフェイズに対する再現率である．

© 2010 Information Processing Society of Japan

再現率は出力フェイズ数が増えるほど向上し，出力フェイズ数が実行履歴中に含まれる機能フェイズ数以上の場合，機能フェイズに対する平均再現率がサブフェイズに対する平均再現率を上回った．

しかし，出力フェイズ数が実行履歴中に含まれる機能フェイズ数未満の場合は，仮説 3 が満たされなかった．これは，機能フェイズの開始点でない 1 つのサブフェイズの開始点が優先して検出されてしまったためである．また，履歴 T-1 における Recall(Feature) は最大で 80%であり，これは履歴 T-1 の 5 つの機能フェイズのうち Logout フェイズで実行されるメソッド呼び出しイベントが特に少なく，1,200 通りのパラメータによる全適用結果すべてにおいてフェイズとして検出されなかったためである．

表 4 より，たとえばツール管理システムの実行履歴から，適当なパラメータ組で本手法を用いて 5 個のフェイズを検出した場合，出力されたフェイズ開始点のうち 93%は開発者が識別したフェイズ開始点と一致し（残り 7%は誤検出であると考えられる），それは開発者が識別した機能フェイズの開始点のうち 56%を含み，開発者が識別したサブフェイズの開始点のうち 39%を含む．また，もし 10 個のフェイズを検出した場合は，そのうち平均 8 個は開発者の識別したフェイズに一致し，残り 2 個は誤検出である．

書籍管理システムの平均適合率は非常に高い値を示したが，正解集合であるフェイズ数が多いために平均再現率は 60%を超えることがなかった．ただし，ある 1 組のパラメータセットで検出されたフェイズに対応する機能は，各実行履歴でほぼ共通していた．実行したシステムの実装が異なるため検出されたサブフェイズは異なっていたが，シナリオ上で共通する機能フェイズの検出結果には大きな差異がみられなかった．この検出結果の安定性は，たとえばデバッグ作業でのバグ修正前と修正後など，実装の詳細が異なるシステムの実行時の振舞いを比較する際に有効である．

4.5 考 察

4.5.1 手法の有効範囲

適用実験では，出力フェイズ数が実行履歴中に含まれる機能フェイズ数以上サブフェイズ数以下である場合，本手法を出力フェイズ数の指定によって用いるための前提条件である 3 つの仮定が満たされることを確認した．

特に機能フェイズ数に近い範囲で出力結果が安定しており，高い適合率と再現率を示している．一方で出力フェイズ数に対するサブフェイズの再現率は低くおさえられており，書籍管理システムの異なる実装のシステムの履歴に対する検出結果の安定性からも，本手法は機能フェイズのような上位の設計に対応する大きなフェイズの検出により有効であるといえる．もしサブフェイズのような実装の詳細に対応する小さなフェイズの検出を行いたい場合には，本手法によって検出された 1 つの機能フェイズに対して再帰的に手法を適用するなどの利用方法が有効である可能性がある．

また，開発者の認識においては粒度の大きい “機能（Feature）” に対応する機能フェイズであっても，その “機能（Feature）” を実現する際に動作するオブジェクトの数や実行されるイベント数が小さい場合，本手法での検出が困難となり，他の機能フェイズを構成するサブフェイズが優先して検出されてしまうことが確認された．このように，本手法は開発者の認識と完全に一致したフェイズ粒度の認識を行うことはできない．機能フェイズとそれを構成するサブフェイズをより厳密に区別した粒度の指定を行うためには，出力フェイズ数だけではなく各機能の実装の詳細に関する知識が必要になると考えられる．

4.5.2 妥当性への脅威

本実験では，実際にシステム開発企業が用いているシステム上で，開発者の作成した実行シナリオを用いた．ただし，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，またシナリオ中の多くの機能フェイズの境界においてユーザの GUI 操作をともなうものである．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．

また，各シナリオの実行時にシステム上で実行されたメソッド呼び出しイベントのうち，Java 標準ライブラリに含まれるクラスのオブジェクトに関するイベントを観測対象から除外している．これは Amida Profiler [21)] の標準の設定であり，オーバヘッドの削減と可視化した際の可読性向上を目的としている．本研究で検出の対象とするフェイズ（機能フェイズ，サブフェイズ）は各フェイズ固有のオブジェクトによって特徴づけられており，それらは対象アプリケーション固有のクラスのオブジェクトである可能性が高く，一方標準ライブラリのような一般的なクラスのオブジェクトは各フェイズに満遍なく登場すると考えられるため，出力フェイズから着目する機能の実行に対応するフェイズを選択する際にはこのような一般的なオブジェクトを除去しておくことは有効であると考えられる．しかし，たとえば String などの小規模なデータオブジェクトはフェイズ内で大量に生成・破棄される一時オブジェクトである可能性が大きく，LRU キャッシュによりフェイズ内で協調動作するオブジェクト群の変化を捕捉する本手法の出力結果の精度に影響を与える可能性がある．プログラムの観測時にすべてのイベントを捕捉することは実行時のオーバヘッドの問題から現実的

© 2010 Information Processing Society of Japan

でないが，どのようなクラスを一般的であるとして除去するかはシステムや解析を行う開発者の目的によって異なる．そこで，多数のソフトウェアで再利用されており開発者にとって既知のものである可能性が高いユーティリティを自動検出する手法[7]などの機械的なフィルタリングを適用することで，実行時のオーバヘッドを抑制しながら本手法の検出結果の精度を向上させることが考えられる．

4.5.3　フェイズと機能の対応付け

本手法の出力は，フェイズの開始点にあたる実行履歴上のイベント時刻の系列である．そのため，開発者は出力されたフェイズがシナリオ上のどのステップに相当するのかを手動で識別しなければならない．しかしこの作業は，開発者が出力フェイズに含まれる機能を示すようなクラスやメソッドなどを手がかりにすることができるため，それほど困難な作業ではないと思われる．

この問題に関連する研究分野としては，feature location や traceability recovery があげられる．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[9]，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[16]．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方法が考えられる．

5. ま と め

本研究では，LRU キャッシュにより協調動作するオブジェクトを観測することで，与えられた実行履歴から自動的にフェイズを検出する手法を提案した．本手法のアルゴリズムは実行履歴情報のみを用いているため，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り，開発者は対象システムに関する詳細な知識なしに利用できる．本手法は我々の開発した実行履歴可視化ツール Amida[⋆1]の一部として実装し，公開している．

今後の課題として，提案手法で検出したフェイズに対し，実行履歴からフェイズごとの特徴を抽出し適切な識別名を与えるなどして，各フェイズと機能名とを自動的に対応付けることがあげられる．また，本手法では固定長の LRU キャッシュを用いたが，システム最適化の分野で提案されている可変長のキャッシュを用いるアプローチ[19]などの適用も考えている．

謝辞　本研究は，文部科学省科学研究費補助金若手研究（B）（課題番号：21700030）の助成を得た．

⋆1 Amida http://sel.ist.osaka-u.ac.jp/˜ishio/amida/

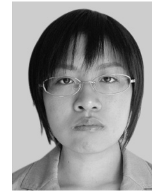
参 考 文 献

1) Briand, L.C., Labiche, Y. and Leduc, J.: Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software, *IEEE Trans. Softw. Eng.*, Vol.32, No.9, pp.642–663 (2006).
2) Chan, K., Liang, L.Z.C. and Michail, A.: Design Recovery of Interactive Graphical Applications, *Proc. 25th International Conference on Software Engineering*, pp.114–124 (2003).
3) Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J.J. and van Deursen, A.: Understanding Execution Traces Using Massive Sequence and Circular Bundle Views, *Proc. 15th International Conference on Program Comprehension*, pp.49–58 (2007).
4) Czyz, J.K. and Jayaraman, B.: Declarative and Visual Debugging in Eclipse, *Eclipse Technology Exchange* (2007).
5) Eisenbarth, T., Koschke, R. and Simon, D.: Locating Features in Source Code, *IEEE Computer*, Vol.29, No.3, pp.210–224 (2003).
6) Hamou-Lhadj, A. and Lethbridge, T.: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System, *Proc. 14th International Conference on Program Comprehension*, pp.181–190 (2006).
7) Ichii, M., Ishio, T. and Inoue, K.: Cross-application Fan-in Analysis for Finding Application-specific Concerns, *Proc. 4th Asian Workshop on Aspect-Oriented Software Development*, pp.39–43 (2008).
8) Ishio, T., Watanabe, Y. and Inoue, K.: AMIDA: A Sequence Diagram Extraction Toolkit Supporting Automatic Phase Detection, *Proc. 30th International Conference on Software Engineering*, pp.969–970 (2008).
9) Koschke, R. and Quante, J.: On Dynamic Feature Location, *Proc. 20th International Conference on Automated Software Engineering*, pp.86–95 (2005).
10) Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol.26, No.6, pp.419–429 (1983).
11) Nagpurkar, P., Hind, M., Krintz, C., Sweeney, P.F. and Rajan, V.T.: Online Phase Detection Algorithms, *Proc. Code Generation and Optimization*, pp.111–123 (2006).
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, *Proc. 23rd International Conference on Software Engineering*, pp.221–230 (2001).

© 2010 Information Processing Society of Japan

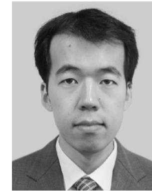
14) Reiss, S.P.: Dynamic Detection and Visualization of Software Phases, *Proc. 3rd International Workshop on Dynamic Analysis*, pp.50–55 (2005).
15) Richner, T. and Ducasse, S.: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles, *Proc. 18th International Conference on Software Maintenance*, pp.34–43 (2002).
16) Rountev, A. and Connell, B.H.: Object Naming Analysis for Reverse-Engineered Sequence Diagrams, *Proc. 27th International Conference on Software Engineering*, pp.254–263 (2005).
17) Salah, M. and Mancoridis, S.: A Hierarchy of Dynamic Software Views From Object-Interactions to Feature-Interactions, *Proc. 20th International Conference on Software Maintenance*, pp.72–81 (2004).
18) Shen, X., Shaw, J., Meeker, B. and Ding, C.: Locality Approximation Using Time, *Proc. 34th Symposium on Principles of Programming Languages*, pp.55–61 (2007).
19) Shen, X., Zhong, Y. and Ding, C.: Locality Phase Prediction, *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.165–176 (2004).
20) Systa, T., Koskimies, K. and Muller, H.: Shimba – An Environment for Reverse Engineering Java Software Systems, *Software – Practice and Experience*, Vol.31, pp.371–394 (2001).
21) 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).
22) Ungar, D.: Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *Proc. Symposium on Practical Software Development Environments*, pp.157–167 (1984).
23) Wang, T. and Roychoudhury, A.: Hierarchical Dynamic Slicing, *Proc. International Symposium on Software Testing and Analysis*, pp.228–238 (2007).
24) Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs, *IEEE Trans. Softw. Eng.*, Vol.18, No.12, pp.1038–1044 (1992).

(平成 22 年 1 月 12 日受付)
(平成 22 年 9 月 17 日採録)



渡邊　結（学生会員）

平成 20 年大阪大学大学院情報科学研究科博士前期課程修了．現在，同大学院情報科学研究科博士後期課程在学中．プログラム解析の研究に従事．



石尾　隆（正会員）

平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年同大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 年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助教．博士（情報科学）．プログラム解析，アスペクト指向ソフトウェア開発に関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．

井上　克郎（フェロー）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59 年〜61 年ハワイ大学マノア校情報工学科助教授．平成 1 年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年大阪大学情報科学研究科コンピュータサイエンス専攻教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

© 2010 Information Processing Society of Japan