プログラム実行履歴を用いた類似クラス・メソッド検出手法

井岡 正和　吉田 則裕　井上 克郎

近年，ソースコードの剽窃が増加している．ソースコードの剽窃は，全体が剽窃される場合と，クラスやメソッド等のソースコードの一部が剽窃される場合がある．ソースコードの一部が剽窃された場合に，剽窃された部分の特定に使用できる技術として，ソフトウェア間に存在する重複部分を特定できるコードクローン検出手法が挙げられる．しかし，ソフトウェアを理解が困難なものに書き換える難読化技術が存在し，剽窃を行った者がソースコードに難読化技術を用いると，剽窃の特定が困難になる．そこで，本研究では，難読化の影響が少ないプログラムの実行履歴を分析することで，類似したクラス対やメソッド対を検出する手法を提案する．提案手法では，実行履歴を複数のフェイズに分割し，各フェイズのメソッド呼び出し列を比較することで，類似したクラス対，メソッド対を検出する．提案手法を実際のアプリケーションに適用した結果，難読化前後で同一のコンポーネントを識別できることを確認できた．

Recently, plagiarism of source code has increased. It can be categorized into whole and partial plagiarisms. For the identification of partial plagiarism, code clone detection techniques can be used because they can detect duplicated parts from two software systems. However, obfuscation techniques allow obscuring plagiarism by rewriting software to make it difficult to understand. In this paper, we propose an approach to detecting pairs of similar classes and methods based on the similarity of execution traces that are resistant to obfuscation. This approach divides an execution traces into phases, and then match them based on the similarity of method calls for the detection of pairs of similar classes and methods. In case study, the approach identified the correspondences of original and obfuscated components in actual applications.

1　はじめに

近年，著作者の意図に反したソースコードのコピーによる再利用等の剽窃が増加している [7] [10]．ソースコードの剽窃は，全体が剽窃される場合と，クラスやメソッド等のソースコードの一部が剽窃される場合がある．ソースコードの一部が剽窃された場合は，ソフトウェア間に存在する重複部分を特定できるコードクローン検出手法を利用して検出することができる [2]．

一方で，ソフトウェアを理解が困難なものに書き換える難読化技術が提案されている [8]．剽窃を行った者が難読化技術を用いると，メソッドが別のクラスに移動される等，プログラムの静的な構造が改変される．そのため，トークン列や構文木の等価性に基づくコードクローン検出手法 [4] では，剽窃を特定することが難しい．また，システム依存グラフ [3] の等価性に基づいてコードクローンを検出することで，難読化に対応可能であると考えられるが，計算コストが非常に大きい．

難読化の影響が少なくソフトウェアの特徴を識別できる情報として，プログラムの実行履歴が考えられる．API 呼び出しの系列や頻度といった実行履歴上の特徴を用いて，ソフトウェア単位の剽窃を検出する手法 [6] が提案されている．しかし，実行履歴からクラスやメソッドの特徴を抽出し，それらの剽窃の特定を行った研究は確認されていない．

そこで本研究では，プログラムの実行履歴からクラ

Detecting Similar Classes and Methods Using Program Execution Traces.
Masakazu Ioka, Katsuro Inoue, 大阪大学, Osaka University.
Norihiro Yoshida, 奈良先端科学技術大学院大学, Nara Institute of Science and Technology.
コンピュータソフトウェア, Vol.31, No.1 (2014), pp.110–115.
[研究論文 (レター)] 2013 年 11 月 19 日受付.

スクローン対，メソッドクローン対 (類似度の高いクラス対，メソッド対) を検出する手法を提案する．提案手法は，2 つのプログラムの実行履歴を与えると，実行履歴を機能的なまとまりであるフェイズに分割し，フェイズ間の類似度を計算して類似度の高いフェイズ対から対応付ける．そして，対応付けられたフェイズ対からクラス間，メソッド間の類似度を計算し，クラスクローン対，メソッドクローン対を出力する．実行履歴分析を行う提案手法は，静的な構造を変化させる難読化だけでなく，Java 言語や C#言語等のリフレクションを生成する難読化にも対応できると考えられる [8].

適用例では，実際のアプリケーションに提案手法を適用し，難読化前後で同一のコンポーネントを識別できることを確認した．

2 背景

2.1 実行履歴の抽出

プログラムの実行履歴とは，実行時のオブジェクトの振る舞いを記録したもので，実行されたイベントが時系列順に並んでいる．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，メッセージの内容等の情報を保持している [11].

プログラムの実行履歴を抽出する方法の 1 つとして Amida [9] がある．Amida のプロファイラ機能は，Java プログラムを動的解析し，実行時のイベントを実行履歴として記録するものである．イベントとしては，メソッドの呼び出し，復帰やフィールドの参照，定義等を検出することができる．

2.2 フェイズ分割手法

我々の研究グループでは，1 つの実行履歴を複数のフェイズに分割する手法を提案している [11]．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」等の開発者にとって意味のある処理に対応するものを指す．

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を持っている [5]．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 Least Recently Used (LRU) キャッシュに登録していくことで，キャッシュの更新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．また，こ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時間コストで計算可能である．

2.3 難読化技術

難読化とは，あるプログラムを理解が困難な実行結果が等価なプログラムに変換することである [8]．難読化を施すことによって，プログラムを不正な解析から保護することができる．一方で，不正にコピーしたプログラムに難読化を施すことによって，剽窃が隠蔽される恐れがある．難読化には，文字列改変を施す名前変換や，構造改変を施すメソッド分散，リフレクションを用いた動的名前解決といった技術が存在する [8].

難読化が施されると，既存のコードクローン検出技術では等価な処理をコードクローンとして識別することが困難である．

3 提案手法

本稿では，2 つのプログラム実行履歴を与えると，その 2 つの実行履歴からクラスクローン対，メソッドクローン対を検出する手法を提案する．この手法では，プログラムの実行履歴を対象としているため，プログラムを難読化された場合でも検出することができる．なお，提案手法ではメソッド呼び出し列を比較するので，ここではプログラムの実行履歴として，「メソッドの呼び出し」と「New 演算 (コンストラクタの呼び出し)」のみをイベントとして用いる．

図 1 に提案手法の概要を示す．提案手法は，以下の手順からなる．

手順 1. フェイズ分割 与えられたプログラムの実行履歴をフェイズに分割する．

手順 2. 正規化 各フェイズのメソッド呼び出しの正規化を行う．

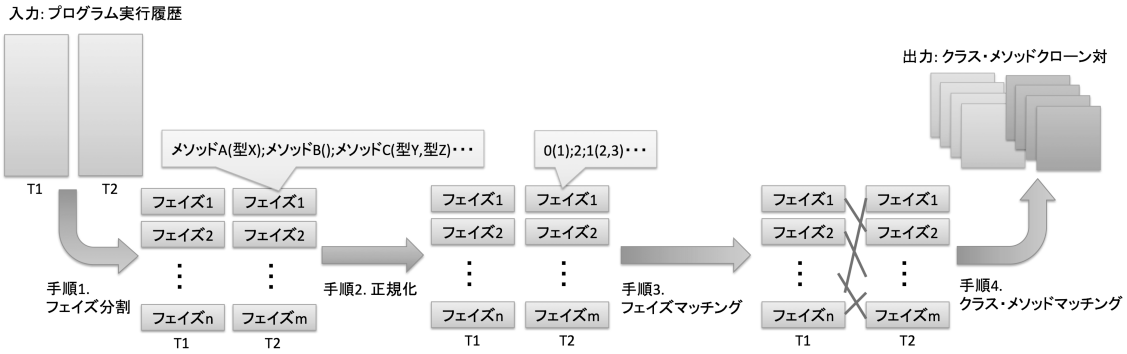

図 1　提案手法の概要

手順 3. フェイズマッチング　動的計画法を用いて各フェイズを比較し対応付ける．

手順 4. クラス・メソッドマッチング　フェイズ間の対応付けを用いてクラスおよびメソッドを対応付ける．

以降，各手順について詳述する．

3.1　フェイズ分割

実行履歴は非常に長く比較の計算コストが大きいので，機能単位での比較を行う．そのために，与えられたプログラムの実行履歴を，2.2 節で挙げたフェイズ分割手法を用いて複数のフェイズに分割する．

3.2　正規化

各フェイズについて 2 種類の正規化を行う．

メソッド呼び出し列の正規化　メソッド呼び出しの繰り返し回数のみが異なる同型のフェイズを類似したフェイズとして検出するため，2 回以上の連続した同じメソッド呼び出しは 2 回のメソッド呼び出しとする．

メソッド呼び出し文の正規化　難読化によってメソッドのシグネチャが意味を持たなくなるため，メソッド呼び出しの系列を，各シグネチャのメソッド呼び出し内の出現順の系列に変換する．その際，1 つ前の呼び出しを出現の起点とすることにより，呼び出し間の関連を考慮する (ただし，フェイズ先頭の呼び出し文は除く)．例を図 2 に示す．呼び出し A(X, Y) については，1 つ前にメソッド呼び出しがないので，呼び出し A(X, Y) 内だけのそれぞれのシグネチャA，X，Y の出現順 0, 1，2 を利用した「0(1, 2);」に変換する．呼び出し B(Z, Z) については，1 つ前のメソッド呼び出しである A(X, Y) を起点として，B，Z の出現順は 3 番目，4 番目となるので「3(4, 4);」となる．呼び出し C(W, Z) についても同様で「2(3, 1);」となる．

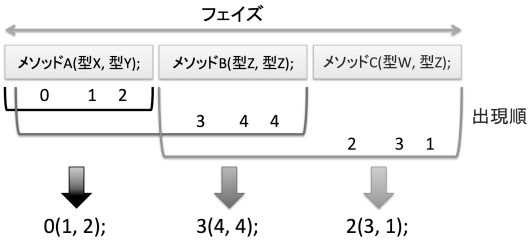

図 2　メソッド呼び出し文の正規化の例

3.3　フェイズマッチング

フェイズの長さ (メソッド呼び出しの数) が短いものは異なる処理のフェイズであってもメソッド呼び出し列が重複しやすいので，フェイズの長さが閾値未満のものを検出対象から除外する．そして，動的計画法を用いた類似文字列マッチングアルゴリズム [12] を使用してフェイズの比較を行う．このアルゴリズムは，入力として 2 つの文字列を与え，一方の文字列に，1 文字削除，1 文字挿入，1 文字置換という 3 つの操作を最低何回か行って，もう一方と同一の文字列へと変化させられるかを調べることで，2 つの文字列の類似度を求めることができる．

提案手法では，出現順に変換した 1 メソッド呼び

出しを 1 文字に符号化してこのアルゴリズムを適用する．このアルゴリズムを用いることで，フェイズ間の類似度と，フェイズ間のメソッド呼び出しの対応関係を得ることができる．なお，フェイズ α 中に出現する全メソッド呼び出しの数を $N_{MC}(\alpha)$，フェイズ α，β 間で対応付けられたメソッド呼び出しの数を $N_{match}(\alpha, \beta)$ とし，フェイズ α とフェイズ β の類似度を以下の式により定義する．

$$\text{類似度 } (\alpha, \beta) = \frac{N_{match}(\alpha, \beta)}{max(N_{MC}(\alpha), N_{MC}(\beta))}$$

2 つの実行履歴の間の全フェイズ間の比較を行った後，フェイズ間の類似度が高いフェイズ対からグリーディに対応付け，得られたフェイズの対を対応するフェイズと呼ぶ．

3.4 クラス・メソッドマッチング

3.3 節で得られた動的なフェイズ間のメソッド呼び出しの対応関係を用いて，静的なクラス A とクラス B の類似度を以下の式により定義する．なお，クラス A 内に出現する全メソッド呼び出しの数を $N_{MC}(A)$，そしてクラス A 内とクラス B 内に現れるメソッド呼び出しのうち，対応するフェイズの対の中において対応付けられたメソッド呼び出しの数を $N_{match}(A, B)$ とする．

$$\text{類似度 (A, B)} = \frac{2 \times N_{match}(A, B)}{N_{MC}(A) + N_{MC}(B)}$$

クラス間の類似度の計算例を図 3 に示す．図 3 の上部にはソースプログラムを，下部にはソースプログラムに対応する実行系列を示している．なお，ソースプログラムには，実行時に出現したメソッド呼び出しのみを記述している．また，それぞれの実行系列 α，β は 2 つのフェイズに分割されており，フェイズ α_1 とフェイズ β_1，フェイズ α_2 とフェイズ β_2 がそれぞれ対応付けられている．例えば，クラス A とクラス B について，クラス A 内のメソッド呼び出し数が 6，クラス B 内のメソッド呼び出し数が 5，クラス A 内のメソッド呼び出しとクラス B 内のメソッド呼び出しが対応付けられている数が 3 であるので，類似度 (A, B) は 0.55 となる．

また，この式のクラス A，クラス B をメソッド A，メソッド B に置き換えた式をメソッド間の類似度として定義する．

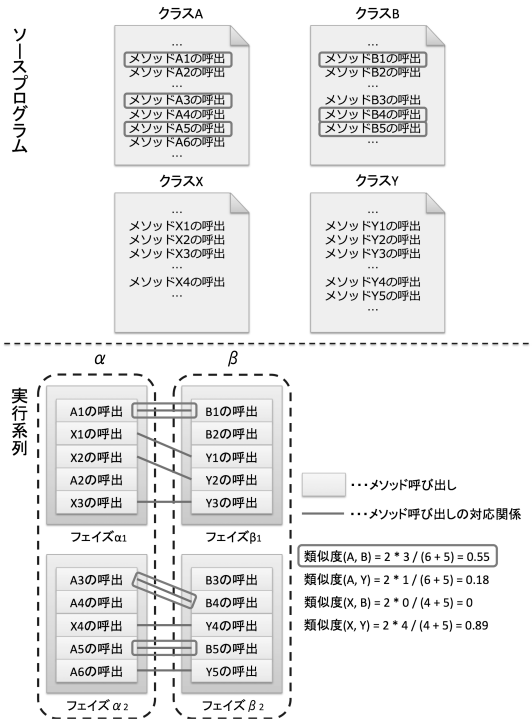

図 3 クラス間の類似度の計算例

2 つの実行履歴の間の全クラス間，全メソッド間の類似度を計算し，類似度が高いものからグリーディに対応付けていき，クラスクローン対，メソッドクローン対 (類似度 (A, B) の高いクラス対，メソッド対) として出力する．

4 適用例

提案手法の難読化に対する耐性を確認するために，提案手法を実装したツールを，実際のアプリケーションに適用した†1．なお，検出するフェイズの長さは 50 以上のみとした．適用対象は ICCA †2 の Gemini コンポーネントと同じく ICCA の Virgo コンポーネントとした．ICCA は統合コードクローン分析環境であり，Gemini コンポーネントはソースコード全体のコードクローン情報を可視化することを目的として開発されており，Virgo コンポーネントはソースコー

†1 フェイズ分割におけるキャッシュサイズ，ウインドウサイズを共に 10 に設定した．

†2 ICCA: http://sel.ist.osaka-u.ac.jp/icca/

ドに含まれるコードクローンに関するメトリクス値を出力することを目的として開発されている．これらコンポーネントに含まれるクラスやメソッドを対象とし，提案手法が難読化前後で同一のクラスやメソッドを識別できるかどうか確認した．

なお，難読化には，Android アプリの盗用対策に標準で用いられている ProGuard†3を難読化ツールの代表的な例として採用し，標準設定で使用した．ProGuard では，2.3 節で述べた名前変換やメソッド分散を用いて難読化を施す．また，各コンポーネントについて，起動からコードクローンデータの解析完了までの実行履歴を取得した．

4.1 クラス・メソッド単位の類似性判定

Gemini コンポーネント，Virgo コンポーネントをそれぞれ ProGuard を用いて難読化し，オリジナルとのクラスおよびメソッドの対応付けを行った†4．それぞれのコンポーネントについて，ProGuard による難読化前後のマッピング情報を用いて，クラスクローン対，メソッドクローン対の対応付けが正しいかどうかを確認した．

結果を表 1 に示す．なお，実行履歴に出現した全クラス (メソッド) 数を S_{all}，提案手法が正しく対応付けしたクラス (メソッド) 数を $S_{propose}$ とする．この結果から，Gemini コンポーネントについては，ほぼすべてのクラスクローン対，メソッドクローン対の対応付けが正しいことが分かる．Virgo コンポーネントについては，すべての対応付けが正しかった．これらのことから，ほぼすべてのクラスおよびメソッドを難読化前後で対応付けできたことが分かる．対応付けが失敗していたクラスやメソッドは，実行履歴中においてそれぞれから高々2 回しかメソッド呼び出しが行われていなかった．そのため，対応関係のないメソッドとの類似度が高くなってしまったメソッドや，任意のクラス (メソッド) との類似度が低くなってしまったクラス (メソッド) が存在し，正しい対応付けを行うことができなかった．

表 1 クラス・メソッドマッチングの結果

	S_{all}	$S_{propose}$	S_{CCF}
Gemini(クラス)	61	59	24
Gemini(メソッド)	73	68	20
Virgo(クラス)	4	4	3
Virgo(メソッド)	4	4	3

†3 http://proguard.sourceforge.net/

†4 Xeon 2.67GHz，2.66GHz のデュアルコアの CPU を備えた計算機上で行った．Gemini コンポーネントについて，フェイズ数は難読化前で 141，難読化後で 92 であり，検出時間は 1 秒以内であった．また，Virgo コンポーネントについて，フェイズ数は難読化前で 116，難読化後で 116 であり，検出時間は 1 秒以内であった．

4.2 コードクローン検出を用いた類似性判定

トークン列の等価性に基づくコードクローン検出ツール CCFinder [4] を用いて，難読化前後のクラス，メソッドをそれぞれ正しく対応付けることができるか調査した．なお，ProGuard での難読化は jar ファイルに対して行うため，Java Decompiler†5を用いて逆コンパイルを行い，難読化後のソースコードを取得した．

結果を表 1 の右端に示す．なお，難読化前後のクラス間，メソッド間でコードクローンが 1 つでも含まれている場合に正解とし，その数を S_{CCF} とおく．この結果より，半数以上のクラス間，メソッド間においてクローンを検出できていないことが分かる．クラス間，メソッド間においてクローンを検出できたものについては，if-else 文の連続等の Java 言語のソースコードから頻繁に検出されるコードクローン [1] であり，同一のクラスおよびメソッドと判定するのは難しいと考えられる．また，提案手法において対応付けに失敗したクラス間，メソッド間からは，コードクローンが検出されなかった．

これらのことから，難読化したプログラムに対して CCFinder を用いた対応付けより，提案手法を用いた対応付けの方が優れていると考えられる．

5 まとめと今後の課題

本稿では，プログラムの実行履歴をフェイズに分割し，各フェイズのメソッド呼び出し列を比較すること

†5 Java Decompiler: http://java.decompiler.free.fr/

で，クラスクローン対，メソッドクローン対を検出する手法を提案した．そして，実際のアプリケーションに提案手法を適用して，難読化前後で同一のコンポーネントを識別できることを確認した．

今後の課題として，他の類似アプリケーションや剽窃の事例に対して実験を行い，正規化手法の改善を行うことが挙げられる．また，実行履歴分析を行う提案手法の有効性は，テストケースの選択方法に依存すると考えられる．そのため，提案手法を使用する際のテストケースの選択方法の考案を今後行う必要がある．

名前変換，メソッド分散以外の代表的な難読化技術であるリフレクションを用いた動的名前解決[8] に対する耐性の確認も今後の課題である．ProGuard がリフレクションを用いた動的名前解決に対応していないため適用例では用いなかったが，実行履歴分析を行う提案手法はこの難読化に対しても高い耐性を持つと期待される．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号: 25220003) の助成を得た．

参 考 文 献

[1] Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and implementation for investigating code clones in a software system, *Information and Software Technology*, Vol. 49, No. 9–10(2007), pp. 985–998.
[2] 肥後芳樹, 楠本真二, 井上克郎：コードクローン検出とその関連技術, 電子情報通信学会論文誌, Vol. J91-D, No. 6(2008), pp. 1465–1481.
[3] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *ACM TOPLAS*, Vol. 12(1990), pp. 26–60.
[4] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7(2002), pp. 654–670.
[5] Lieberman, H. and Hewitt, C.: A real-time garbage collector based on the lifetimes of objects, *CACM*, Vol. 26, No. 6(1983), pp. 419–429.
[6] 岡本圭司, 玉田春昭, 中村匡秀, 門田暁人, 松本健一：API 呼び出しを用いた動的バースマーク, 電子情報通信学会論文誌, Vol. J89-D, No. 8(2006), pp. 1751–1763.
[7] Raymond, E. and Landley, R.: OSI position paper on the SCO-vs.-IBM complaint, 2004, http://www.catb.org/~esr/hackerlore/sco-vs-ibm.html.
[8] 玉田春昭, 中村匡秀, 門田暁人, 松本健一：Java クラスファイル難読化ツール DonQuixote，日本ソフトウェア科学会 FOSE 2006, 2006, pp. 113–118.
[9] 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(2007), pp. 153–169.
[10] Ueno, T.: Pocketmascot に対する抗議ページ, 2001, http://www.tomozon.sakura.ne.jp/wince/About_PocketMascot/About_PocketMascot.html.
[11] 渡邊結, 石尾隆, 井上克郎： 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, 情報処理学会論文誌, Vol. 51, No. 12(2010), pp. 2273–2286.
[12] Yates, R. B. and Neto, B. R.: *Modern Information Retrieval*, Addison Wesley, 1999.

井 岡 正 和

2011 年大阪大学基礎工学部情報科学科卒業．2013 年同大学大学院博士前期課程修了．現在，新日鉄住金ソリューションズ株式会社に勤務．在学中，リファクタリング支援および実行履歴分析の研究に従事．

吉 田 則 裕

2004 年九州工業大学情報工学部知能情報工学科卒業．2009 年大阪大学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)．2010 年奈良先端科学技術大学院大学情報科学研究科助教．博士 (情報科学)．コードクローン分析手法やリファクタリング支援手法に関する研究に従事．

井 上 克 郎

1984 年大阪大学大学院基礎工学研究科博士後期課程修了 (工学博士)．同年，大阪大学基礎工学部情報工学科助手．1984〜1986 年，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991 年大阪大学基礎工学部助教授．1995 年同学部教授．2002 年大阪大学大学院情報科学研究科教授．2011 年 8 月より大阪大学大学院情報科学研究科研究科長．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．