複数プログラミング言語で記述されたソフトウェアからのコードクローン検出

中村 勇太[1,a)]　崔 恩瀞[2]　吉田 則裕[3]　春名 修介[1]　井上 克郎[1]

概要：複数のプログラミング言語 (以降，言語) で記述されたソフトウェアでは，異なる言語を組合わせて1つの機能を実装するため，各言語で記述されたソースコードは互いに呼出し関係を持つ．異なる言語で記述されたソフトウェアから検出されたコードクローンは呼出し関係を持つことが多く，呼出し関係で結合された，より大きなコード片に対して新しくコードクローンを定義することができる．このコードクローンは呼出し関係の前後で実装している機能は同じだと考えられるため，互いに一致もしくは類似した機能を実装していると考えられる．このようなコードクローンを開発者に提示することにより，ライブラリ作成支援やリファクタリングなどが機能単位で行え，コードクローンの保守作業がより効率的になると期待できる．しかし，既存のコードクローン検出手法では呼出し関係を考慮していないため，複数のコードクローン検出結果を見比べて呼出し関係を把握するという作業が必要である．そこで本研究では，このコードクローンを InterLanguage Clone (ILC) として定義し，その自動検出手法について提案する．さらに，実際のオープンソースの複数言語ソフトウェアに対してケーススタディを実施し，検出される ILC が同一もしくは類似した機能を実装しているかどうか調査した．その結果，そのようなケースがいくつか存在することを確認できた．

キーワード：コードクローン検出，ソフトウェア保守，複数言語ソフトウェア

Detection of Software Clones Written in Multiple Programming Languages

Yuta Nakamura[1,a)]　Eunjong Choi[2]　Norihiro Yoshida[3]　Syusuke Haruna[1]　Katsuro Inoue[1]

1. はじめに

コードクローンとは，ソースコード中に存在する互いに一致もしくは類似した部分を持つコード片のことであり，主にソースコードのコピーアンドペーストによって生成される [1], [2].

ソースコード中のコードクローンには様々な活用方法がある．例えば，既存のソースコードから再利用可能なライブラリを作成する際は，頻繁に再利用されているコード片を把握する必要があり，その把握にコードクローンが有用である．また，コードクローンはソフトウェア保守に悪影響を及ぼすという指摘もされている [3], [4], [5]．コードクローンを含むソフトウェアの保守性を改善するためには，ソースコード中のコードクローンを 1 つのモジュールに集約するリファクタリングを検討しなければならない [6]．コードクローンはソフトウェア開発や保守の支援に活用されるため，開発者はソースコード中から見つける必要がある．しかし，開発者が開発のたびに目視でコードクローンを見つけることは開発における時間的な制約を考えると現実的ではない．したがって，コードクローンを自動で検出し，開発者に提示する支援が必要となる．

[1] 大阪大学
Osaka University
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[3] 名古屋大学
Nagoya University
[a)] n-yuuta@ist.osaka-u.ac.jp

© 2016 Information Processing Society of Japan

これまで，様々なコードクローン自動検出手法の提案やその適用実験が行われてきた [4], [7], [8]．しかし，これらの研究は，主に単一のプログラミング言語 (以降，言語) で記述されたソフトウェアを対象にしているため，複数の言語を組合わせてより高度なソフトウェアが開発されてきている現状にあってない [9]．さらに，複数の言語で記述されたソフトウェア (以降，複数言語ソフトウェア) にも正しく適用できない恐れがある．

複数言語ソフトウェアの場合，異なる言語を組合わせて 1 つの機能を実装する．例えば，ウェブアプリケーションである機能を実装する際は，クライアント側の実装とサーバ側の実装で異なる言語を用いる．そのため，各言語で記述されたソースコードは互いに呼出し関係を持つ．その結果，各言語で記述されたソースコードから検出されたコードクローン同士が呼出し関係を持つことがある．

例えば，コード片 $(C_i, C_{i'})$ と $(C_j, C_{j'})$ がコードクローンであり，$C_i \to C_j$ や $C_{i'} \to C_{j'}$ というように呼出し関係を持ち，さらに呼出し元と呼出し先が異なる言語で記述された場合，この呼出し関係で結合された，より大きなコード片に対して新しくコードクローンを定義することができる．つまり，$(C_i \to C_j)$ と $(C_{i'} \to C_{j'})$ が新しいコードクローンとなる．呼出し関係の前後で実装している機能は同じだと考えられるため，この新しいコードクローン同士は互いに一致もしくは類似した機能を実装していると考えられる．したがって，一致もしくは類似した機能を実装しているコードクローンを開発者に提示することで，ライブラリ作成支援やリファクタリングといったコードクローンの活用がより効率的になると期待できる．

しかし，既存のコードクローン検出手法は複数言語ソフトウェアを想定してない上，呼出し関係を考慮していないため，既存のコードクローン検出手法で複数言語ソフトウェアからコードクローンを検出する場合，複数のコードクローン検出結果を見比べて呼出し関係を把握する必要があり，作業効率が下がってしまう．

そこで本研究では，このコードクローンを InterLanguage Clone (ILC) として定義し，その自動検出手法について提案する．さらに，実際のオープンソースの複数言語ソフトウェアに対してケーススタディを実施し，検出される ILC が一致もしくは類似した機能を実装しているかどうか調査した．

以降，2 章では本研究で提案する ILC の定義について述べ，3 章ではその検出手法を説明する．そして 4 章ではケーススタディとその結果について述べ，5 章では関連研究を紹介する．最後に 6 章で本研究のまとめと今後の課題について述べる．

2. Interanguage Clone

コードクローンとは，互いに一致もしくは類似した部分を持つコード片であり，主にソースコードのコピーアンドペーストによって生成される [1], [2]．一般に，互いにコードクローンであるコード片のペアをクローンペア，そしてコードクローンの同値類をクローンセットと呼ぶ．

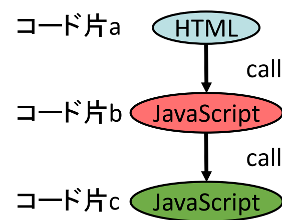

図 1　呼出し関係グラフ (CRG).
Fig. 1　Call Relation Graph.

複数言語ソフトウェアは複数の言語の特性を組合わせて機能を実装する．その際，異なる言語で記述されたソースコード間に呼出し関係が生じる．このような複数言語ソフトウェアからコードクローンを検出した場合，検出されたコードクローンが互いに呼出し関係を持つことが多いある．本研究では，この呼出し関係で結合された，より大きなコード片に対して定義される新しいコードクローンを Interlanguage Clone (ILC) と定義する．以降，ILC の詳しい定義について説明する．

ILC を定義するにあたり，まず呼出し関係グラフを定義する．コード片の集合が与えられたときに，コード片間の呼出し関係に基づいて有向グラフを構築することができる．図 1 は呼出し関係グラフ (CRG) の例を表している．このグラフのノードはコード片を，ノード間に引かれるエッジはコード片間の呼出し関係を表している．また，各ノードに付けられた名前はそのコード片を記述している言語を表す．図 1 では，HTML で記述されたコード片 a が JavaScript で記述されたコード片 b を呼出し，さらにそのコード片 b が同じく JavaScript で記述されたコード片 c を呼出している．このような有向グラフを呼出し関係グラフ (CRG) と定義する．

呼出し関係グラフに基づいて，ILC を次のように定義する．与えられた 2 つのコード片集合 1 と 2 に対して，呼出し関係グラフ CRG1 と CRG2 を構築する．これら 2 つのグラフ CRG1 と CRG2 が以下の条件を満たすとき，与えられたコード片集合 1 と 2 は互いに ILC となる．

(1) グラフが複数の言語で記述されたコード片で構築されている．
(2) グラフが同型である．
(3) グラフの対応する各ノード同士がコードクローンの関係にある．

図 2 は 2 つのコード片集合から構築される呼出し関係グラフの例を表している．この図において，2 つのコード片集合から構築された呼出し関係グラフ CRG1 と CRG2 は同じ型をしており，対応するノード同士がコードクローン

© 2016 Information Processing Society of Japan

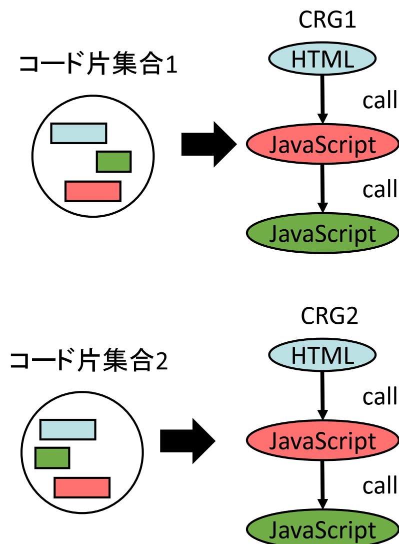

図 2　2 つのコード片集合から構築される呼出し関係グラフ．同色のノード同士は互いにコードクローンの関係にあることを表す．
Fig. 2　Call Relation Graphs Constructed from Two Code Fragment Sets. Two Nodes which Have the Same Color Mean that They are Code Clone.

の関係にある．さらに，グラフは HTML と JavaScript で記述されたコード片で構築されている．したがってこの場合，2 つのコード片集合は互いに ILC となる．

さらに，通常のコードクローンにおけるクローンペアとクローンセットを ILC の場合でも同様に定義することができる．つまり，互いに ILC の関係にあるコード片集合のペアが ILC ペア，そして ILC の同値類が ILC セットとなる．

3. Interlanguage Clone 検出手法

この章では，2 章で定義した ILC の自動検出手法について説明する．検出手法は 3 つのステップで構成されている (図 3)．

ステップ 1： 入力ソースコードに対して既存のコードクローン検出ツールを適用し，各言語毎にコードクローンを検出する

ステップ 2： 検出されたコードクローンに対して静的解析を行い，コードクローン間の呼出し関係情報を抽出する．入力ソースコード全体ではなく，検出されたコードクローンを対象として静的解析を行うのは，解析に掛かる時間を削減するためである．

ステップ 3： 検出されたコードクローンをノード，抽出した呼出し関係情報をエッジとする呼出し関係グラフを構築し，定義に基づいて ILC を検出する．

以降，ステップ 3 についてその詳細を述べる．

ステップ 3 の ILC 検出は，構築された呼出し関係グラフ全体の中から，

- 複数の言語を用いている．
- 同型である．
- 対応する各ノードがコードクローンの関係にある．

以上 3 つの条件を満たす部分グラフのペアを見つけることに等しい．その手順は以下の通りである．

手順 1： 同じクローンセット内に存在する任意のコードクローンを 2 つ選択する．

手順 2： 選択したコードクローンそれぞれに対して，呼出し先/呼出し元のコードクローンを取得する．

手順 3： 呼び出し先/呼出し元のコードクローンが共に同じクローンセットに含まれていれば，同型部分グラフ候補として追加する．

手順 4： 呼び出し先/呼出し元が存在しない場合，同じクローンセットに含まれていない場合，既に同型部分グラフ候補に追加済みの場合は探索を終了する．

この処理を再帰的に行うことで，対応するノードがコードクローンの関係にあるような同型部分グラフのペアを得ることができる．その中で，複数の言語が用いられていれば，その同型部分グラフのペアを ILC ペアとして検出する．手順 4 における，追加済みかどうかの条件は呼出し関係が循環している場合に探索が無限ループに陥ってしまう事態を防ぐために設けた．

4. ケーススタディ

4.1 概要

本研究では複数言語ソフトウェアを対象として，ILC の定義とその自動検出手法を提案した．その目的は，1 章で述べたように，複数の言語で記述されたソースコード間で呼出し関係を持たせることにより機能を実装している複数言語ソフトウェアから，同一もしくは類似した機能を実装しているコードクローンを検出することである．

そこで，実際のオープンソースソフトウェアを用いたケーススタディを実施し，ILC を検出することでこの目的を実現できているかどうかを確認した．ケーススタディでは以下の 2 点について調査を行った．

(1) ILC はどの程度存在するのか．
(2) I 各 LC セット内の全 ILC は同一もしくは類似機能を実装しているか．

以降，ケーススタディの対象と調査方法，そしてその結果や考察について述べる．

4.2 対象

ILC は複数言語ソフトウェア全般に対して定義することができる．しかし今回のケーススタディでは，特定のドメ

表 1　対象ウェブアプリケーション (WA) の規模．
Table 1　Statistics of Target Web Applications.

WA 名	総行数 (HTML)	総行数 (JavaScript)
Webogram	6,146	224,140
DuckieTV	3,404	67,485

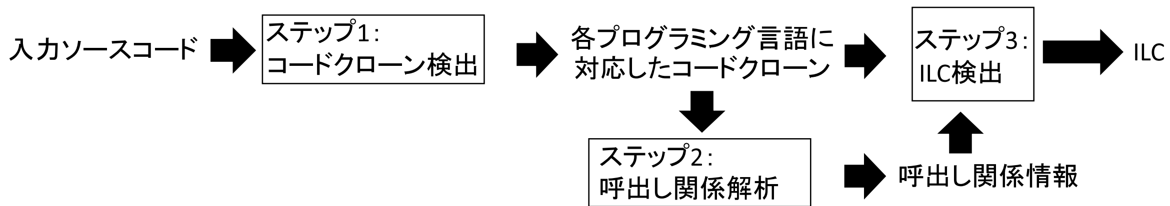

図 3 ILC 検出手法の概要.
Fig. 3 Overview of Proposed ILC Detection Approach.

イン，及び言語に焦点を当てた．これは，ドメインや言語を絞ることによりコードクローン検出や呼出し関係解析の精度を上げるためである．

今回のケーススタディでは，ドメインについてはウェブアプリケーションを，そして言語については HTML と JavaScript を選択した．ウェブアプリケーションは，リリース期間の短さやパフォーマンス低下回避などが要因でコードクローンを多く含むことが分かっている．[10], [11]. ILC の検出には通常のコードクローンが必要となるため，コードクローンが多いウェブアプリケーションからは十分な数の ILC が検出されると考えた．また，過去の研究で GitHub*1 上のリポジトリを調査した結果，複数の言語を用いて開発が行われていた 66,825 リポジトリの内，22.9%のリポジトリで HTML と JavaScript を組合わせて開発を行っていたことが分かった [12]．これは，他の利用言語の組み合わせの中でも最多であった．よって，対象となるソフトウェアを探しやすいと考えて，HTML と JavaScript を利用言語に選んだ．

さらに，十分な規模のウェブアプリケーションを選択するための条件として，(1) コミット数が 1000 以上，(2) 人気度を表すスターが 100 個以上付けられたウェブアプリケーションのみに絞った．そしてその中から，無作為にウェブアプリケーションを選択し，今回の対象とした．

表 1 に，今回のケーススタディで用いた 2 つのウェブアプリケーション (Webogram*2，DuckieTV*3) に関するデータを示した．表中の値はそれぞれ，HTML ファイルの総行数と JavaScript ファイルの総行数を表している．Webogram はチャットアプリである Telegram をウェブアプリケーション化したものである．また，DuckieTV は，カレンダー上でドラマなどのテレビ番組の更新情報を管理するためのウェブアプリケーションである．

4.3 調査方法

ILC はどの程度存在するのか.

3 章で説明した検出手法にしたがって，2 つのウェブアプリケーションから ILC を検出する．コードクローンを検出するステップでは，既存のコードクローン検出である NiCad[4] を用いた．NiCad は行単位でソースコードを比較し，その類似度が指定した閾値以上であるときにコードクローンであると判定する．NiCad は，文法情報と，関数やブロックといった検出粒度を与えることで，任意の言語から任意の粒度でコードクローンを検出することができる．専用の検出ツールが存在しない HTML と JavaScript からコードクローンを検出するにあたり，文法情報を与えるだけで検出が行える拡張性の高さが理由で NiCad を選択した．

NiCad に与えるパラメータは，検出粒度，類似度の閾値，そして検出されるコードクローン行数の閾値がある．今回のケーススタディでは，HTML に対するコードクローン検出の場合，検出粒度を開始タグから終了タグまでの一括り，類似度の閾値を 0.3，最小行数を 10 行とした．また，JavaScript に対するコードクローン検出の場合，検出粒度を関数，類似度の閾値を 0.3，最小行数を 5 行とした．さらに，文法構造が同じでもユーザ定義名に違いがあるコード片をコードクローンとして検出するために，どちらの場合もユーザ定義名をすべて同一トークンに変換している．

呼出し関係を解析するステップでは，HTML 内に存在する関数呼び出しを HTML→JavaScript の呼出し関係，JavaScript 内の関数呼び出しを JavaScript→JavaScript の呼出し関係として抽出した．

こうして検出されるコードクローンと，それらの間の呼出し関係情報を用いて ILC を検出し，検出された ILC セットの数を確認する．

各 ILC セット内の全 ILC は同一もしくは類似機能を実装しているか.

検出された各 ILC セット内の全 ILC が実装している機能を調べ，それらが同一もしくは類似した機能かどうかを判断する．すべての ILC が同一もしくは類似した機能を実装している場合に限り，その ILC セットは同一もしくは類似した機能を実装したコードクローンが集まっていると判断する．

4.4 結果

ILC はどの程度存在するのか.

2 つのウェブアプリケーションから検出された ILC セット

*1 https://github.com/
*2 https://github.com/zhukov/webogram
*3 https://github.com/SchizoDuckie/DuckieTV

© 2016 Information Processing Society of Japan

表 2　検出された ILC セットの数及び，HTML と JavaScript から検出されたクローンセットの数

Table 2　Number of Detected ILC Sets and Original(HTML&JavaScript) Clone Sets.

WA 名	ILC セット数	HTML クローンセット数	JavaScript クローンセット数
Webogram	4	97	44
DuckieTV	3	11	65

表 3　各 ILC セット内の ILC が同一もしくは類似機能を実装しているかどうかの判断結果.

Table 3　The Result of Judging Whether All the ILCs of Each ILC Set Implement the Same or Similar Functionalities or not.

WA 名	ILC セット番号	ILC 数	同一もしくは類似機能を実装しているか (Y/N)
Webogram	1	2	N
	2	2	Y
	3	2	N
	4	3	Y
DuckieTV	1	2	Y
	2	2	Y
	3	2	Y

の数について**表 2** にまとめた．Webogram に関して，元々は HTML 単体で 97 個のクローンセットが，JavaScript 単体で 44 個のクローンセットが検出されていた．これに対して ILC 検出を行ったところ，4 個の ILC セットが検出された．また，DuckieTV の場合は，HTML 単体で 11 個のクローンセットが，JavaScript 単体では 65 個のクローンセットが検出されていた．これに対して ILC 検出を行うと，3 個の ILC セットが検出された．

各 ILC セット内の全 ILC は同一もしくは類似機能を実装しているか．

検出された各 ILC セット内の全 ILC が同一もしくは類似機能を実装しているかどうか判断した結果を**表 3** に示した．Webogram については，検出された 4 個の ILC セットの内，2 個の ILC セット内の全 ILC が類似機能を実装していた．DuckieTV は，3 個の ILC セットすべてにおいて，ILC セット内の全 ILC が類似機能を実装していた．

図 4 に，Webogram から検出された ILC セット内の全 ILC が，類似機能を実装していたケースにおける実際のソースコードの例を，図 5 に，同じく Webogram から，異なる機能を実装していたケースにおけるソースコードの例を載せた．

図 4 はどちらも，チャットアプリ内に存在する様々な切り替えボタン (トグル) を押したときの処理を実装しているため，類似機能を実装していると判断した．一方，図 5 の例では，上のコード片がタイトルの編集という処理を，下のコード片がチャット内のチャンネルに参加する際の処理を表している．これらは異なる機能を実装していると判断した．

4.5　考察

表 3 より，ILC セットを構成している ILC がすべて類似機能を実装しているケースが検出されている．したがって，開発者に ILC 検出結果を提示することで，類似機能がまとまっていることを活かしたコードクローン利用を行うことができると考えられる．例えばライブラリ作成を行う場合にコードクローンの情報を用いる場合は，全く異なる機能を実装しているコードクローンを提示される場合に比べて効率よくライブラリ候補を探すことができる．

表 2 から分かるように，元々検出されていたコードクローンの大部分が ILC セットとして検出されなくなっている．これは，呼出し関係を持たないコードクローンは ILC として検出されなくなってしまうからである．このことから，ILC 検出結果を見るだけでは本来利用価値のあった重要なコードクローンまでも無視してしまう恐れがあることが分かる．そこで，開発者にはまず ILC 検出結果を見てもらい，開発時間に余裕があればその後，既存のコードクローン検出ツールによる結果を見て細部にまで目を向けてもらうという 2 段階の利用方法が考えられる．ウェブアプリケーションのように，リリース期間が短く，時間的制約が厳しい開発環境においては，既存のコードクローン検出と ILC 検出を組合わせるこの利用方法が有用であると考えられる．

今回のケーススタディでは，ILC セット内の全 ILC が類似機能を実装していないケースがみられた (表 3)．その例を見てみると (図 5)，どちらも言語の構文的構造は類似しているものの，使われている識別子が異なっていることが分かる．したがって，検出される ILC セット内の全 ILC が同一もしくは類似した機能を実装しているようにするためには，こうした識別子に着目して手法を改善する必要が

© 2016 Information Processing Society of Japan

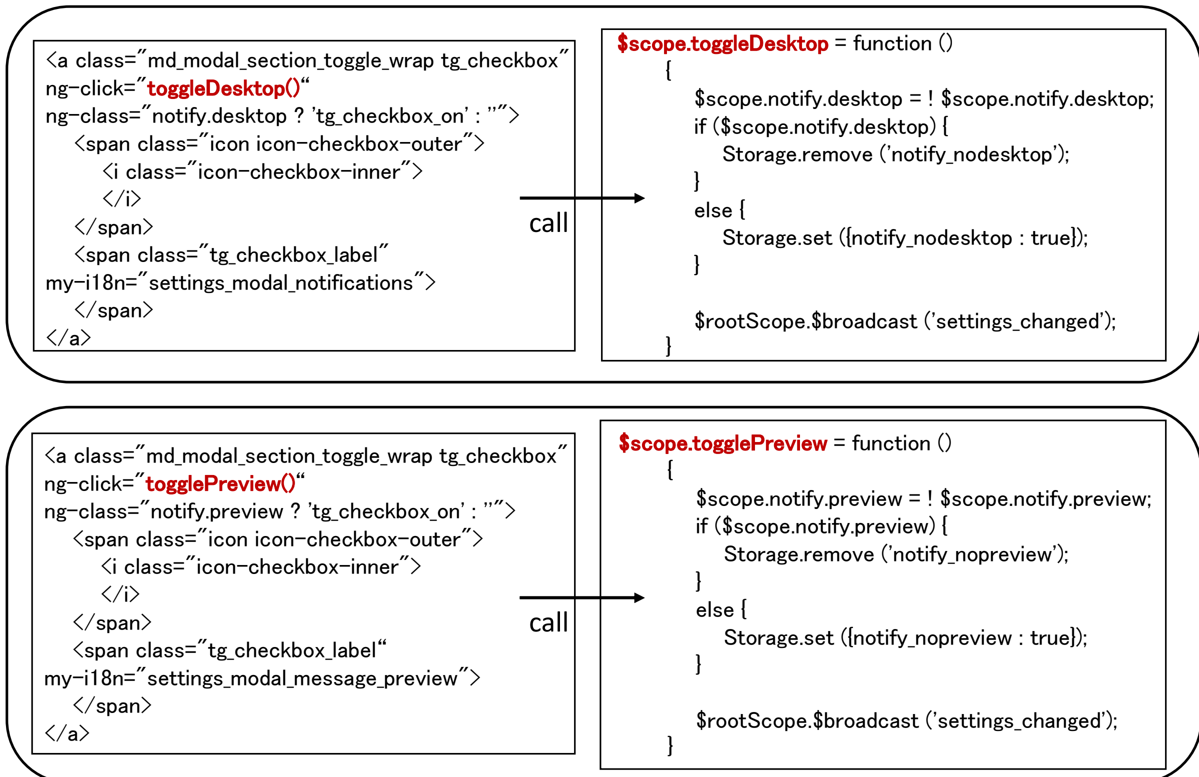

図 4　類似機能を実装している例．赤字は呼出し関係部分．

Fig. 4　Example of ILCs which Implement Similar Functionalities.

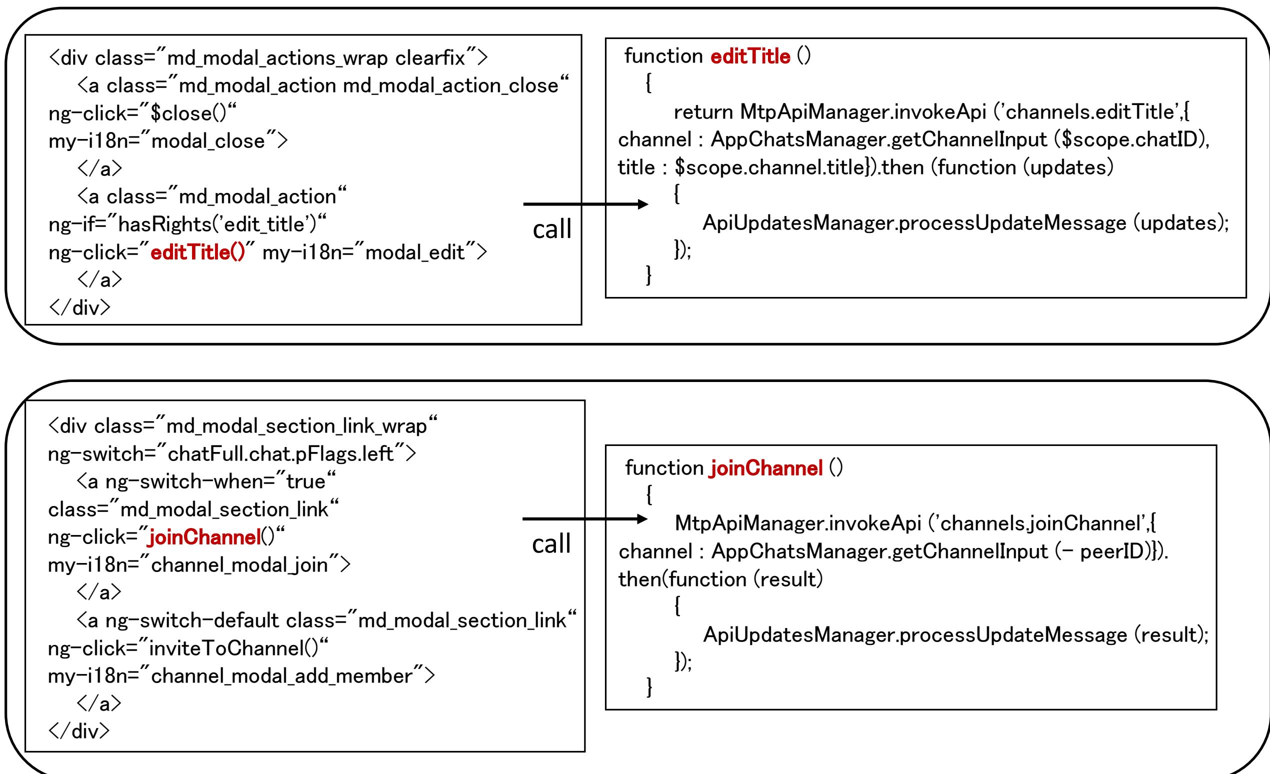

図 5　異なる機能を実装している例．赤字は呼出し関係部分．

Fig. 5　Example of ILCs which Implement Different Functionalities.

© 2016 Information Processing Society of Japan

ある．例えば，関数呼出し文に使われている識別子が大きく異なる場合にはILCとして検出しないという方法が考えられる．

5. 関連研究

単一のプログラミング言語で記述されたソフトウェアからコードクローンを検出する様々なツールが提案されてきた．Kamiyaらはトークンベースのコードクローン検出ツールCCFinderを開発した[7]．CCFinderは，字句解析を行うことによってソースコードをトークン列に変換し，変数名や関数名などのユーザ定義名を同一トークンに変換する．その後，閾値以上の長さの共通トークン列を探索することによって，コードクローンを検出する．Jiangらはソースコードの構文構造を木構造で表した抽象構文木を用いてコードクローンを検出するツールDECKARDを開発した[8]．DECKARDは構文解析を行うことによってソースコードを抽象構文木に変換し，類似した部分木を探索することによってコードクローンを検出する．RoyらはTextベースのコードクローンツールNiCadを開発した[4]．NiCadはTXL's agile parsing[13]を用いてソースコードを正規化した後に，最長共通部分列アルゴリズムを用いてコードクローンを検出する．

複数のプログラミング言語で記述されたソフトウェアからコードクローンを検出する手法も提案されてきた．KraftらはC#とVisual Basic.NET言語間でのコードクローンを検出する手法を提案した[3]．彼らの手法はソースコードをCodeDOM(Code Document Object Model)グラフに変換し，トークン列を抽出した後に，レーベンシュタイン距離[14]に基づいてコードクローンを検出する．Chengらはgitのログから C#と Javaプログラミング言語間でのdiffクローンを検出するツールCLCMinerを開発した[15]．CLCMinerはgitのログを解析し，diffの列やそれぞれdiffの属性（コミット日付，コミットＩＤなど）を抽出する．抽出されたdiffはトークン列に正規化され，同じファイル名のdiffの類似度に基づいてコードクローンを検出する．

ウェブアプリケーションを対象にコードクローンを検出し，調査した研究がある．RajapakseらはCCFinderを用いて17のウェブアプリケーションを対象にコードクローンを検出し調査した[10]．その結果，ウェブアプリケーションに含まれるソースコードのうち17-63%がコードクローンとなっていることが分かった．また，彼らはCCFinderを用いてウェブアプリケーションのサーバページを対象にコードクローン検出し，実証研究を行った[11]．その結果，ほとんどのサーバページのコードクローンは集約可能であり，コードクローンの集約によるソースコードサイズの減少などの利点がある反面，パフォーマンス低下などの欠点がありトレードオフの関係にあることが分かった．

Nguyenらは複数のプログラミング言語で記述された動的なwebアプリケーションを対象にしたスライシング手法WebSliceを提案し，5つのシステムを対象にした実証実験を行った[16]．その結果，プログラミング言語を横断するデータフローやスライスを多数確認できた．最後に，Muhammadらは動的ウェブページからisland grammar[17]を用いてPHPソースコードを抽出した後，NiCadを用いてコードクローンを検出し，ウェブアプリケーションにおけるPHPコードクローンを調査した[18]．

上述の通り，複数言語ソフトウェアを対象としたコードクローンの検出やその調査は行われてきたが，いずれもコードクローン間の呼出し関係に着目していない．複数言語ソフトウェアから検出されたコードクローン間に存在する呼出し関係に着目して新たなコードクローン(ILC)を検出したのは本研究が初めてである．

6. まとめと今後の課題

複数言語ソフトウェアでは，異なる言語で記述されたソースコード間に呼出し関係が存在する．そのため，異なる言語で記述されたソースコードに含まれるコードクローン同士も呼出し関係を持つ．このとき，呼出し関係で結合されたコードクローンの組をより大きな1つのコード片とみると，このコード片に対して新しくコードクローンが定義できる．本研究ではこれをILCと定義し，さらに自動検出手法を提案した．

複数言語ソフトウェアの1つであるウェブアプリケーションに対してケーススタディを実施した結果，2つの対象ウェブアプリケーションから計7個のILCセットを検出した．さらに，各ILCセット内の全ILCが同一もしくは類似した機能を実装しているケースが5個確認できた．このことから，ILC検出を行うことで類似機能を実装したコードクローンが1つにまとまることに繋がると結論を出した．この状態で開発者に結果を提示することで，例えば類似機能をまとめてライブラリを作成したり，機能単位でコードクローンの集約を効率よく行うことが可能になると考えられる．

今後の課題としては，検出に識別子の情報を組み込むことで，ILCセット内の全ILCが類似機能を実装しているような検出結果を出すことが挙げられる．また，ソフトウェア開発や保守の観点から，既存のコードクローンよりもILCを開発者に提示することで本当により効率的に作業が行えるかどうかを検証していきたい．

謝辞 本研究はJSPS科研費JP25220003，JP26730036，JP15H06344，JP16K16034の助成を受けたものです．

参考文献

[1] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees,

© 2016 Information Processing Society of Japan

Proc. of ICSM 1998, pp. 368–377 (1998).

[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 (ソフトウェア工学)，電子情報通信学会論文誌. D, 情報・システム， Vol. 91, No. 6, pp. 1465–1481 (2008).

[3] Kraft, N., Bonds, B. and Smith, R.: Cross-Language Clone Detection, *Proc. of SEKE 2008*, pp. 54–59 (2008).

[4] Chanchal, K. R. and James, R. C.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *Proc. of ICPC 2008*, pp. 172–181 (2008).

[5] 吉田則裕，肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローン間の依存関係に基づくリファクタリング支援，情報処理学会論文誌，Vol. 48, No. 3, pp. 1431–1442 (2007).

[6] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison Wesley (1999).

[7] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code., *IEEE Trans. Software Eng.*, Vol. 28, No. 7, pp. 654–670 (2002).

[8] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. of ICSE 2007*, pp. 96–105 (2007).

[9] Baishakhi, R., Daryl, P., Vladimir, F. and Premkumar, D.: A Large Scale Study of Programming Languages and Code Quality in Github, *Proc. of FSE 2014*, pp. 155–165 (2014).

[10] Rajapakse, D. C. and Jarzabek, S.: An Investigation of Cloning in Web Applications, *Proc. of ICWE 2005*, pp. 252–262 (2005).

[11] Rajapakse, D. C. and Jarzabek, S.: Using Server Pages to Unify Clones in Web Applications: A Trade-Off Analysis, *Proc. of ICSE 2007*, pp. 116–126 (2007).

[12] Nakamura, Y., Choi, E., Yoshida, N., Haruna, S. and Inoue, K.: Towards Detection and Analysis of Interlanguage Clones for Multilingual Web Applications, *Proc. of IWSC 2016*, pp. 17–18 (2016).

[13] Dean, T. R., Cordy, J. R., Malton, A. J. and Schneider, K. A.: Agile Parsing in TXL, *Automated Software Engg.*, Vol. 10, No. 4, pp. 311–336 (2003).

[14] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady*, Vol. 10, pp. 707–710 (1966).

[15] Cheng, X., Peng, Z., Jiang, L., Zhong, H., Yu, H. and Zhao, J.: Mining Revision Histories to Detect Cross-language Clones Without Intermediates, *Proc. of ASE 2016*, pp. 696–701 (2016).

[16] Nguyen, H. V., Kästner, C. and Nguyen, T. N.: Cross-language Program Slicing for Dynamic Web Applications, *Proc. of ESEC/FSE 2015*, pp. 369–380 (2015).

[17] Synytskyy, N., Cordy, J. R. and Dean, T. R.: Robust Multilingual Parsing Using Island Grammars, *Proc. of CASCON 2003*, pp. 266–278 (2003).

[18] Muhammad, T., Zibran, M. F., Yamamoto, Y. and Roy, C. K.: Near-miss Clone Patterns in Web Applications: An Empirical Study with Industrial Systems, *Proc. of CCECE 2013*, pp. 1–6 (2013).

© 2016 Information Processing Society of Japan