修士学位論文

題目

リファクタリング支援を目的とした言語横断の依存関係を利用したコードクローン検出法

指導教員
井上 克郎 教授

報告者
中村 勇太

平成 29 年 2 月 7 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 28 年度 修士学位論文

リファクタリング支援を目的とした言語横断の依存関係を利用したコードクローン検出法

中村 勇太

内容梗概

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，互いにコードクローンの関係にあるコード片の集合はクローンセットという．一般的に，コードクローンの存在はソフトウェアの保守を困難にするといわれている．そこで，ソフトウェアの保守性を改善するために，リファクタリングを実施することが推奨されている．リファクタリングとは，ソフトウェアの外部的振る舞いを変化させることなくソフトウェアの内部的構造を改善する技術であり，コードクローンに対しては類似コード片を 1 つのモジュールにまとめる集約を実施する．

リファクタリングを実施するためには，ソースコード中のコードクローンの存在を把握する必要がある．そこで，これまでにコードクローンの自動検出手法についての研究がされてきた．しかし，これらの研究は単一の言語で記述されたソフトウェアを対象としており，複数の言語で記述されたソフトウェアを対象とはしていない．現在は複数の言語を組合わせてソフトウェアの開発がされることが珍しくないため，こうしたソフトウェアからのコードクローン検出についても考えるべきであるが，既存研究は複数の言語を利用していることを考慮していないため，複数言語ソフトウェアに特化したコードクローン検出とはいえない．

複数の言語で記述されたソフトウェアでは，1 つの機能を実装するために複数の言語を用いることがある．そのため，各言語で記述されたソースコード間で呼出し関係やデータの共有といった依存関係を持つ．その結果，検出されるコードクローン間にも，言語横断の依存関係が存在する．このことから，言語横断の依存関係を持つコードクローンをまとめた集合は，類似機能を実装したコードクローンのみで構成された集合となっていることが期待できる．一般的に，リファクタリングは類似した機能単位に対して実施されることから，依存関係を持つコードクローンの集合をまとめて開発者に提示することで，開発者はリファクタリングの候補になりやすいコードクローンをチェックすることが可能となる．しかし，検出されたコードクローン間の言語横断の依存関係を把握しておくことは開発者にとって困難な作業である．

吉田らは，こうした依存関係を持つコードクローンをチェーンドクローン，そしてその集合をチェーンドクローンセットとして定義し，チェーンドクローンセットを利用したリファ

クタリング支援手法を提案した．しかしこの手法は単一言語内の依存関係のみに着目しているため，言語横断の依存関係を含む複数言語ソフトウェアに対して有効な手法ではない．

そこで本研究では，言語横断の依存関係を持つコードクローンを言語間チェーンドクローン，そしてその集合を言語間チェーンドクローンセットとし定義し，それらを検出する手法を提案する．さらに，複数言語で記述されたウェブアプリケーションを対象にケーススタディを実施した．その結果，言語間チェーンドクローンの存在，そして類似機能を実装したコード片のみで構成された言語間チェーンドクローンセットの存在を確認できた．

主な用語

コードクローン
依存関係解析
リファクタリング支援
ウェブアプリケーション

目 次

1　まえがき

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 [3, 18]．また，互いにコードクローンの関係にあるコード片の集合はクローンセットという．

一般的に，コードクローンの存在はソフトウェアの保守を困難にするといわれている [5, 12, 17]．例えば，コードクローンに対して修正を加えた際は，同一クローンセット内に含まれる他のコードクローンに対しても同様の修正が必要かどうかを検討しなければならない．

ソフトウェアの保守性を改善するためには，リファクタリングの実施が考えられる．リファクタリングとは，ソフトウェアの外部的振る舞いを変化させることなくソフトウェアの内部的構造を改善する技術である [8]．Fowler は，リファクタリングを実施するべきソースコードの特徴をまとめており，その中の 1 つに重複コード (コードクローン) を挙げている．また，コードクローンに対するリファクタリングパターンとして集約を紹介している．集約とは，同一クローンセット内のコードクローンを 1 つのモジュールにまとめることであり，集約を行うことによってコードクローンが除去され，ソフトウェアの保守性が改善される．

リファクタリングを実施するためには，ソースコード中のコードクローンの存在を把握しておく必要がある．しかし，ソースコードの規模が大きくなるにつれコードクローンも多くなるので，開発者が目視でその存在を把握することは困難な作業である．そこで，これまでに様々なコードクローン自動検出手法が研究されてきた [5, 10, 11]．その一方で，これらの研究は単一の言語で記述されたソフトウェアを対象としており，複数の言語で記述されたソフトウェアを対象とはしていない．現在は複数の言語を組合わせてソフトウェアの開発がされることは珍しくないため [2]，こうしたソフトウェアからのコードクローン検出についても考えるべきであるが，既存研究は複数の言語を利用していることを考慮していないため，複数言語ソフトウェアに特化したコードクローン検出とはいえない．

複数の言語で記述されたソフトウェアでは，1 つの機能を実装するために複数の言語を用いることがある．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間で呼出し関係やデータの共有といった依存関係を持つ．その結果，検出されるコードクローン間にも，言語横断の依存関係が存在する．このことから，言語横断の依存関係を持つコードクローンをまとめた集合は，類似機能を実装したコードクローンのみで構成された集合となっていることが期待できる．一般的に，リファクタリングは類似した機能単位に対して実施されることから，依存関係を持つコードクローンの集合をまとめて開発者に提示することで，開発者はリファクタリングの候補になりやすいコードクローンをチェックすることが可能となる．しかし，検出されたコードクローン間の言語横断の依存関係を把握して

おくことは開発者にとって困難な作業である．

過去の研究で吉田らは，このような依存関係を持つコードクローンをチェーンドクローン，そしてその集合をチェーンドクローンセットとして定義し，チェーンドクローンセットを利用したリファクタリング支援手法を提案している [17]．しかし，彼らの手法は対象を単一言語 (Java) に限定しているため，言語を横断する依存関係を含む複数言語ソフトウェアに対して有効な手法ではない

そこで本研究では，言語横断の依存関係を持つコードクローンを言語間チェーンドクローン，そしてその集合を言語間チェーンドクローンセットとして定義し，それらを検出する手法を提案する．さらに，複数言語で記述されたウェブアプリケーションを対象にケーススタディを実施した．その結果，言語間チェーンドクローンが存在すること，そして類似機能を実装したコードクローンのみで構成された言語間チェーンドクローンセットの存在を確認した．

以降，2 章では研究の背景を説明し，3 章では本研究で定義する言語間チェーンドクローンセットについて述べる．さらに 4 章で今回実施したケーススタディとその結果について説明する．最後に，5 章で本研究のまとめと今後の課題について述べる．

2　背景

この章では本研究の背景として，コードクローンとコードクローンに対するリファクタリング，そしてチェーンドクローンについて説明する．

2.1　コードクローン

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る [3, 18]．また，コードクローンの関係にあるコード片の組をクローンペア，互いにコードクローンの関係にあるコード片の集合をクローンセットという．

コードクローンには厳密で普遍的な定義は存在しないが，Bellon らはコードクローン間の違いの度合に基づき，コードクローンを以下の 3 つに分類している [4]．

タイプ 1

インデントや括弧の付け方などのコーディングスタイルやコメントの有無を除いて完全に一致する．

タイプ 2

タイプ 1 のコードクローンに加えて変数名，関数名などのユーザ定義の識別子や変数，関数などの型が異なる．

タイプ 3

タイプ 2 のコードクローンに加えて，文の挿入や削除，変更が行われている．

コードクローンは主にソースコードのコピーアンドペーストなどの既存のソースコードを再利用することにより，開発効率を向上させるために生成される．[3, 18]．しかし，コードクローンの存在はソフトウェアの保守性を悪化させる要因の 1 つであるとされている [5, 12, 17]．例えば，ソースコード中のあるコード片に修正を加えた場合，修正を加えたコード片とコードクローンの関係にあるコード片に対しても同様の修正が必要となる可能性がある．コードクローンに対して一貫した修正が行われていないとバグの原因になる恐れがあるため，開発者は同一クローンセット内のコードクローン全てに対して修正が必要かどうかを検討しなければいけなくなる．

コードクローンによるソフトウェア保守性の低下を防ぐためには，ソースコード中のコードクローンを見つけ，除去する必要がある．しかし，ソースコードの規模が大きくなると開発者がコードクローンの存在を把握・管理しておくことが困難になる．そこで，これまでに様々なコードクローンの自動検出手法やツールが提案されてきた [5, 10, 11]．以降，2.2 節ではコードクローン検出ツール NiCad について説明し，2.3 節ではコードクローンを除去する手段であるリファクタリングについて述べる．

2.2 コードクローン検出ツール: NiCad

NiCad は入力として与えられたソースファイル集合からコードクローンを検出し，その結果を HTML 形式や XML 形式で出力するコードクローン検出ツールである [5]．NiCad は，ソースコード変換のための言語 TXL を利用してコードクローンを検出する [7]．TXL スクリプトは，図 1(a) のように文脈自由文法形式で記述された文法定義部とソースコードを変換するためのルールを記述したルール定義部で構成されている．入力ソースコードが与えられると，TXL は定義された文法に従ってソースコードをパースし，構文木として内部保持する．次に，定義された変換ルールに従って構文木中の要素を変換し，構文木からソースコードを再度構築して出力する．

TXL は標準でいくつか機能を持っている．例えば，文法定義部で [IN]，[]NL]，[EX] を記述することでソースコードの出力形式を指定することができ，それぞれインデントの設定，改行の挿入，インデントの解除を表している．これらの機能を使ったソースコードの整形を pretty-print という．また，ルール定義部で [^] とプログラム要素を併せて記述することでソースコードからそのプログラム要素を抽出することができる．図 1 中の (b) と (c) は TXL スクリプトによってソースコードがどのように変換，出力されるかを示した例である．入力ソースコード (b) はルール「renameIds」によって識別子 [id] が記号 x に変換される．その後，if_statement の文法定義に従って，インデントや改行が入った状態 (c) で出力される．

こうしたソースコード変換や抽出，pretty-print 機能を備えている TXL を用いて，NiCad は次の 3 つのステップでコードクローンを検出する．

ステップ 1: コードクローン候補の抽出

入力ソースコードからコードクローンの候補となるコード片集合を抽出する．デフォルトでは NiCad は関数単位もしくはブロック単位でコード片を抽出する．さらにこのステップでは文法定義部の記述に従って，抽出したコード片に対して pretty-print が適用される．

ステップ 2: ソースコードの正規化

必要があれば，抽出したコードクローンの候補となるコード片に対して正規化を行う．NiCad は正規化として rename，abstraction，filtering の 3 つの操作を提供している．rename はコード片中の識別子をすべて同一記号に変換する操作である．また，abstraction はプログラム要素を変換する操作で，例えば関数呼び出しの引数全体を一括で Arguments というプログラム要素に変換することで引数の数の違いを吸収することができる．最後に，filtering はプログラム要素を除去する操作である．宣言文や初期化文はソースコードのロジックに大きく影響しないことから，コードクローン検出の際に除去するべきであり，その処理に filtering を適用するという利用方法が考えら

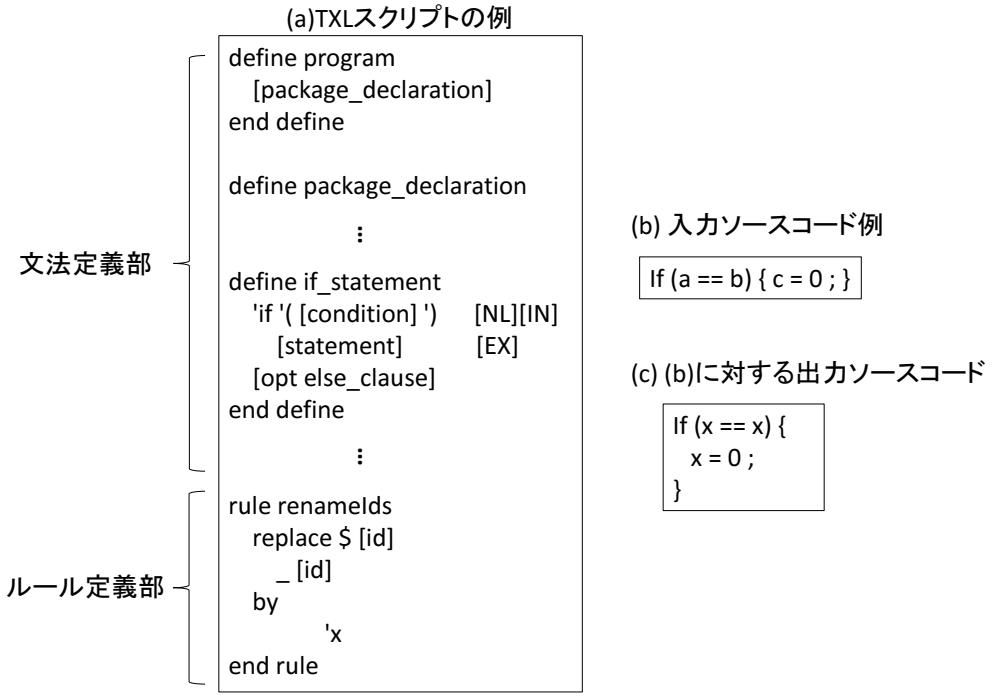

図 1: TXL スクリプトとソースコード変換の例．(a) は入力ソースコードの文法定義と変換ルールを記述したスクリプト例で，(a) を適用することで入力ソースコード (b) はソースコード (c) に変換，出力される．

れる．

ステップ 3: コードクローン検出

ステップ 1 とステップ 2 によってコード片の抽出，pretty-print の適用，正規化が行われた．ステップ 3 ではコード片同士を行単位で比較しコードクローンを見つける．具体的には，2 つのコード片の最長共通部分列 [9] を求め，それを基に 2 つのコード片の不一致行を見つける．そしてコード片の全行数に対する不一致行数の比率が設定した閾値以下である場合に 2 つのコード片をコードクローンとして出力する．例として，図 2 のコード片に対して閾値 0.3 という設定の下，比較を行ったとする．このとき，不一致行は 3 行目のみで，全行数に対する不一致行数の比率が 1/4=0.25 と 0.3 を下回る．従って，このケースではコード片 a とコード片 b はコードクローンとして検出される．

以上のような TXL の性質と NiCad の検出ステップを踏まえると，NiCad は次のような特徴を持つ．

コード片a	コード片b	一致(○) / 不一致(×)
1: for (i=0; i<10; i++) {	1: for (i=0; i<10; i++) {	○
2: a=b+c;	2: a=b+c;	○
3: System.out.println(a);	3: show(a);	×
4: }	4: }	○

図 2: NiCad によるコード片比較の例

任意の単位でのコードクローン検出が容易

コードクローンの候補となるコード片の抽出は，文法として定義されているプログラム要素を TXL プログラム中の変換ルール部分に指定することで行える．そのため，ブロックやファイル全体，関数など様々な検出単位への切り替えが容易である．

タイプ 1 からタイプ 3 までのコードクローン検出が可能

上述の通り，ステップ 1 で pretty-print を適用するため入力ソースコードのコーディングスタイルが統一される．よって，タイプ 1 のコードクローンを検出することができる．また，正規化ステップにおいて rename を適用することでタイプ 2 のコードクローンを検出することができる．さらに，比較ステップにおける閾値を変更することでタイプ 3 のコードクローンを検出することができる．

任意の言語への対応が容易 NiCad のコードクローン検出手順の中で言語に依存しているのは TXL を用いるステップ 1 とステップ 2 である．そのため，文法定義や変換ルールを記述したスクリプトを各言語毎に用意することで任意の言語で記述されたソースコードからのコードクローン検出が可能となる．

2.3 コードクローンに対するリファクタリング

ソフトウェアの保守性を改善するためにリファクタリングの実施が推奨されている．リファクタリングとは，ソフトウェアの外部的振る舞いを変化させることなくソフトウェアの内部的構造を改善する技術である [8]．Fowler はリファクタリングを実施するべきソースコードの特徴をまとめており，その中の 1 つに重複コード (コードクローン) を挙げている．また，コードクローンに対するリファクタリングパターンとして集約を紹介している．集約とは，クローンセット内のコードクローンを 1 つのモジュールにまとめることであり，集約を実施することによってコードクローンが除去され，ソフトウェアの保守性が改善される．

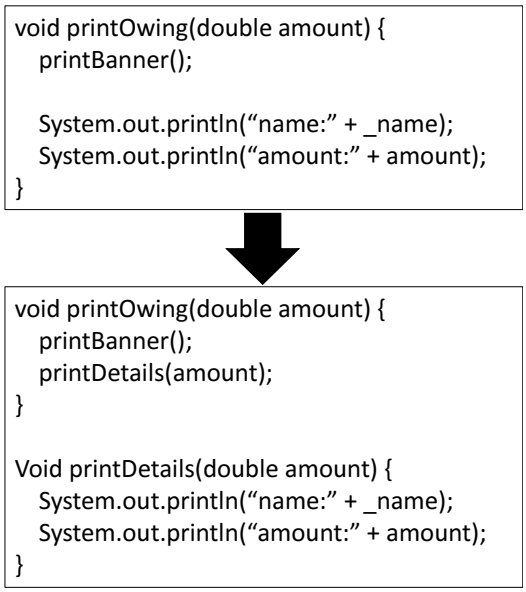

図 3: メソッドの抽出

また，コードクローンに対するリファクタリングは類似多機能単位で実施されることが一般的である．以下では，Fowler が自身の著書の中で紹介している集約方法の例をオブジェクト指向プログラミング言語の 1 つである Java を用いて説明している．

メソッドの抽出

同一クラス内の複数メソッドに同じ式がある場合，同一部分を抽出した後，新たなメソッドと定義し各メソッドの内部から抽出したメソッドを呼び出す (図 3)．

メソッドの引き上げ

同じ結果をもたらすメソッドが複数のサブクラスに存在する場合は，それらをスーパークラスに引き上げる (図 4)．

テンプレートメソッドの形成

異なるサブクラスの 2 つのメソッドが，類似の処理を同じ順序で実行しているが，各処理は異なっている場合がある．このとき，元のメソッドが同一になるように，各処理を同じシグニチャを持つメソッドにし，メソッドの引き上げを行う (図 5)．

2.4 チェーンドクローン

コードクローン検出ツールの検出結果をもとにリファクタリングを実施することでコードクローンを除去し，ソフトウェアの保守性を改善することができる．しかし，コードクロー

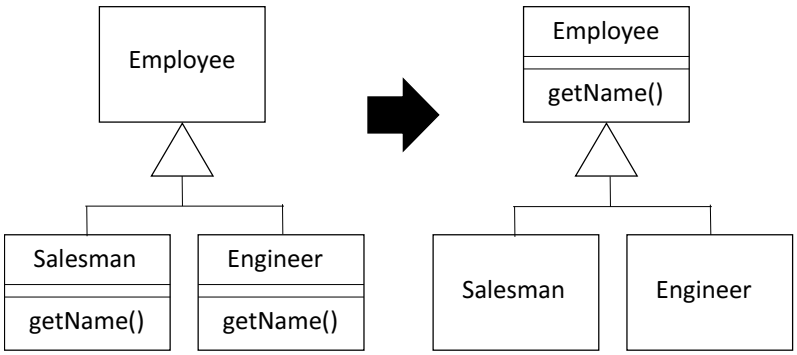

図 4: メソッドの引き上げ

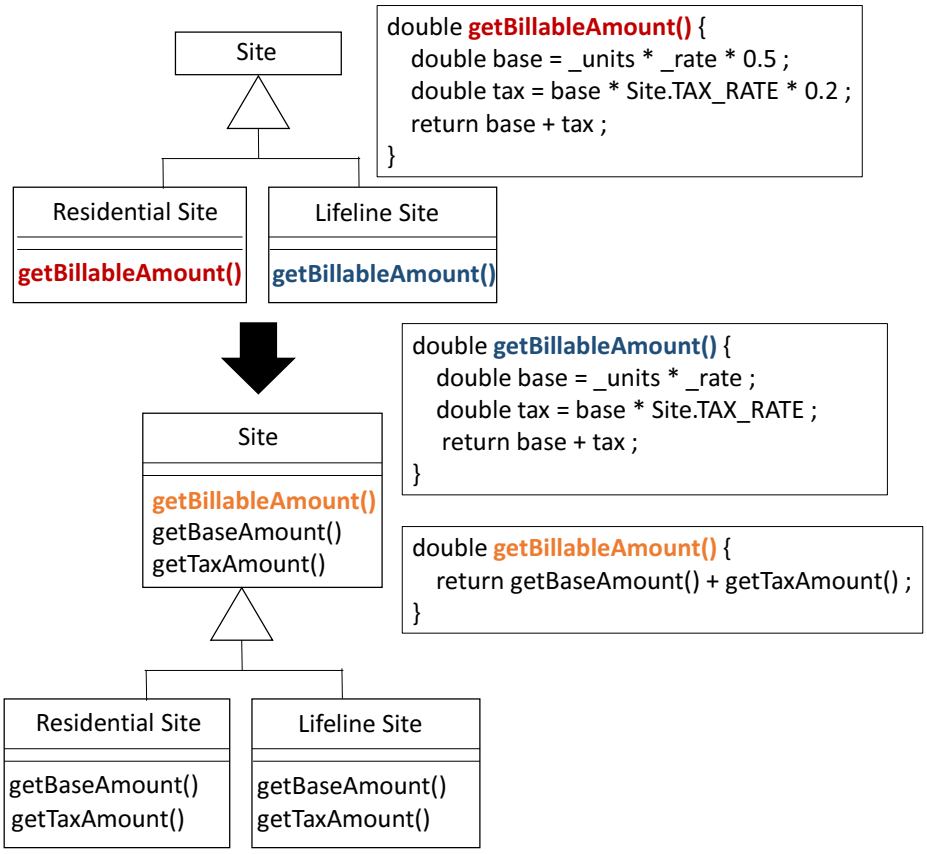

図 5: テンプレートメソッドの形成

ンに対するリファクタリングの適用が困難になる場合が存在する．それは，コードクローンが依存関係を持つ場合である．例として，図 6 はクラス A には 2 つのメソッド F1 と F2 があり，メソッド F1 はメソッド F2 を呼出している．また，兄弟クラス B にも 2 つのメソッド

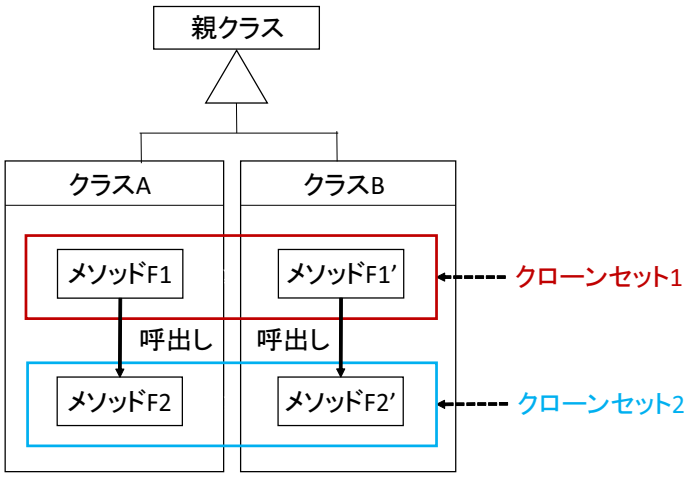

図 6: 依存関係を持つコードクローンの例

F1' と F2' があり，同様にメソッド F1' からメソッド F2' を呼出している．さらに，メソッド F1 とメソッド F1' は同一クローンセットに属するコードクローンであるとする．このように依存関係 (呼出し関係) を持つコードクローンに対してメソッド引き上げリファクタリングを実施することができる．この場合，単純にメソッド引き上げを実施すると，図 7 のようにメソッド F1 とメソッド F1' を集約して作成されたメソッド newF1 から子クラスのメソッド F2 と F2' を呼出すことが不可能になってしまう．リファクタリングが実施された後も，実施前 (図 6) と同等な依存関係を持たせるためには，図 7 において親クラスに抽象メソッドを追加する必要がある．しかし，もし依存先 (呼び出し先) もコードクローンになっていることが分かれば (図 6 クローンセット 2)，図 8 のように 2 つのクローンセットに対して同時にリファクタリングを実施し，集約後のメソッド newF1 からメソッド newF2 へ依存関係 (呼出し関係) を繋ぐのみで済む．

このように，コードクローン同士が依存関係を持つ場合には依存先と依存元のクローンセットに対して同時にリファクタリングを実施することでリファクタリング作業が容易になる．しかし，開発者がコードクローン検出ツールの結果を基にコードクローン同士の依存関係を把握しておくことは難しいため，開発者はより困難なリファクタリング作業をしなければいけない．そこで，依存関係を持つコードクローン集合をまとめて提示することで開発者のリファクタリング作業を支援することができる考えられる．

吉田らの研究では，依存関係を持つコードクローンをチェーンドクローン，さらにその集合をチェーンドクローンセットとして定義した (図 9)[17]．さらに，チェーンドクローンセットを提示することがリファクタリング支援として有効かどうかを調査した．調査の結果，チェーンドクローンセットの規模が通常のクローンセットと比較して大きいこと，及び

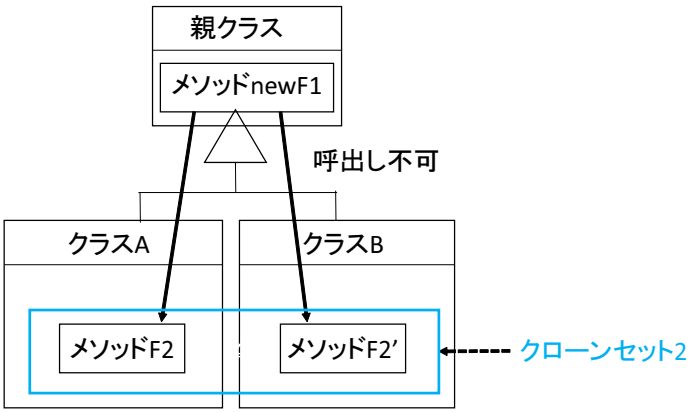

図 7: 図 6 の状態から，クローンセット 1 のみにメソッド引き上げリファクタリングを実施した結果

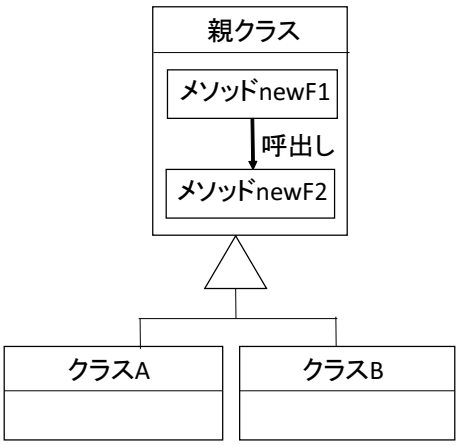

図 8: 図 6 の状態から，クローンセット 1 とクローンセット 2 に同時にメソッド引き上げファクタリングを実施した結果

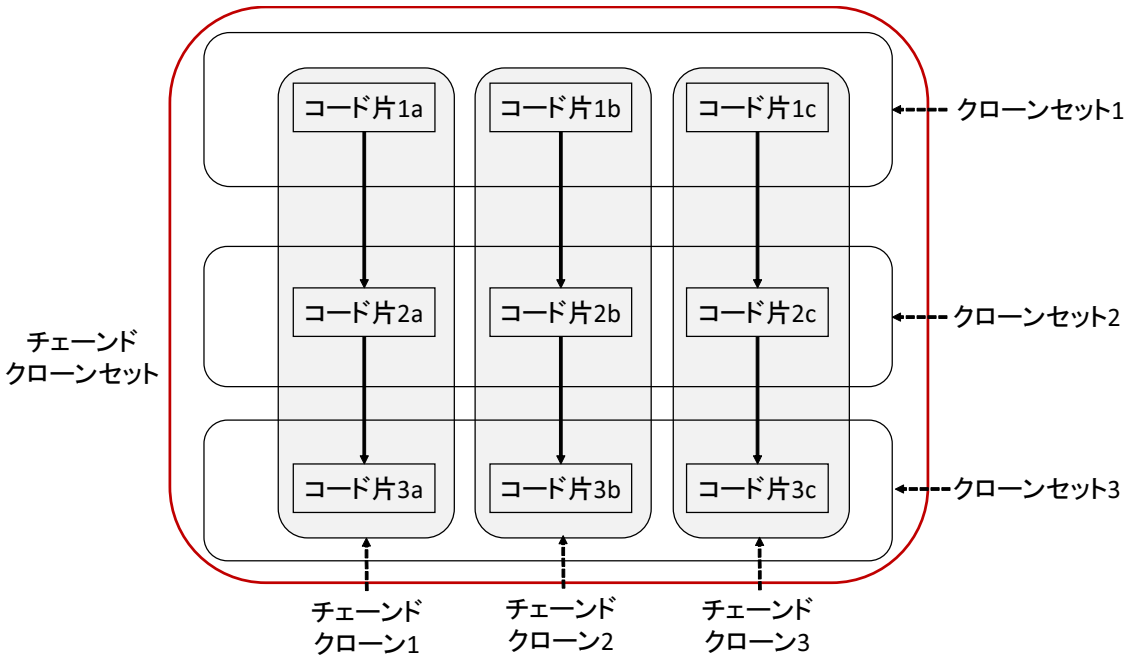

図 9: チェーンドクローンとチェーンドクローンセット

チェーンドクローンセットに対するリファクタリングがクローンセット単位のリファクタリングと比較して容易であることから，チェーンドクローンセットの提示がリファクタリング支援に有効であることが分かった．

3 提案手法

3.1 研究動機

現在のソフトウェア開発では，複数の言語を組合わせて開発を行うことが珍しくなく [2]，複数言語ソフトウェアにおいても，単一言語ソフトウェアと同様にコードクローンが存在する．しかし，2 章で述べたような既存のコードクローン検出やコードクローンの利用に関する研究は単一言語ソフトウェアを対象としていることがほとんどである．複数言語を対象としている研究も存在するが，複数言語で構成されたソースコードの中から単一言語で記述されたソースコードを抽出した上でのコードクローン検出であったり [13]，異なる言語で記述されたソースコード間でのコードクローンを検出する [12] など，複数言語を利用していることを考慮したコードクローン検出とはいえない．そこで，複数言語を利用していることに着目したコードクローン検出について考える．

複数言語ソフトウェアの場合，複数の言語を組合わせて 1 つの機能を実装することがある．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間で呼出し関係やデータの共有といった依存関係を持つ．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間にも依存関係が存在することになる．このことから，言語横断の依存関係を持つコードクローンをまとめた集合は，類似機能を実装したコードクローンのみで構成された集合となっていることが期待できる．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っている場合，依存関係の前後を合わせて 1 つの機能を実装していると考えられるため，$C_i \to C_j$ と $C_{i'} \to C_{j'}$ は類似した機能を実装していると考えられる．一般的に，リファクタリングは類似した機能単位に対して実施されることから，依存関係を持つコードクローンの集合をまとめて開発者に提示することで，開発者はリファクタリングの候補になりやすいコードクローンをチェックすることが可能となる．しかし，検出されたコードクローン間の言語横断の依存関係を把握しておくことは開発者にとって困難な作業である．

依存関係を持つコードクローンをまとめて開発者に提示するということは 2.4 節でも説明した通り，吉田らが既に行っている．しかし，彼らの研究は単一言語 (Java) 内に限定した依存関係に着目しているため，言語横断の依存関係を持つ複数言語ソフトウェアに対しては有効ではない．そこで本研究では，言語横断の依存関係を持つコードクローンを言語間チェーンドクローン，そしてその集合を言語間チェーンドクローンセットとして定義し，それらを検出する手法を提案する．

3.2 言語間チェーンドクローン

3.2.1 定義

複数言語 $(L_1, L_2, \ldots, L_n)$ で記述されたソフトウェアから検出された，各言語に対応するクローンセット集合内に含まれるコード片 (コードクローン) 集合の和集合を $CF = \{CF_{L_1}, CF_{L_2}, \ldots, CF_{L_n}\}$ とする．また，各コード片間に存在している依存関係の種類を表したラベル集合を L，依存関係集合を D とする．このとき，コード片集合 CF を頂点集合，依存関係集合 D を辺集合，依存関係の種類を表した集合 L をラベル集合とするラベル付き有向グラフ G を構築することができる．ここで，ラベル集合 L に含まれる要素は以下の通りである．

- C_{inter}：関数呼出し (言語間).
- C_{inner}：関数呼出し (言語内).
- R_{inter}：同一変数の参照 (言語間)
- R_{inner}：同一変数の参照 (言語内)
- A_{inter}：同一変数への代入 (言語間)
- A_{inner}：同一変数への代入 (言語内)

さらに，言語を横断する依存関係を表すラベル集合を L_{inter} とする．すなわち $L_{inter} = \{C_{inter}, R_{inter}, A_{inter}\}$ である．また，ラベル付き有向グラフ G に含まれる 2 つの部分グラフを G1，G2 とし，対応するコード片集合を CF_{G1}, CF_{G2}，そのラベル集合を L_{G1}，L_{G2} とする．このとき，2 つの部分グラフ G1 と G2 に対して，

1. グラフが同型である.
2. 対応する各頂点が表すコード片同士がコードクローンの関係にある.
3. 対応する辺に付けられているラベルが同一である.
4. $|L_{G1} \cap L_{inter}| \geq 1$.
5. $|L_{G2} \cap L_{inter}| \geq 1$.

以上の条件が成立する場合，コード片集合は互いに言語間チェーンドクローンとなる．また，言語間チェーンドクローンに対する同値類を言語間チェーンドクローンセットと定義する．

図 10 は言語間チェーンドクローンとなる 2 つのコード片集合とそれらから構築される部分グラフの例を表している．どちらも同型の部分グラフであり，対応する各頂点同士がコード

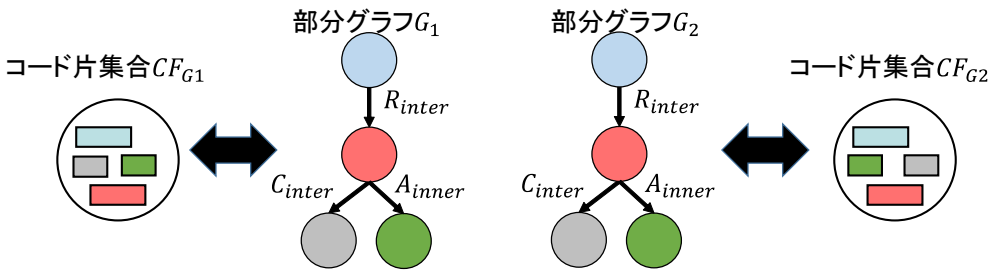

図 10: 2 つのコード片集合から構築される部分グラフが言語間チェーンドクローンとなる条件を満たす例．同色の頂点は互いにコードクローンの関係にあることを表す．

クローンとなっている．また，対応する辺に付けられているラベルが同一であり，言語横断の依存関係を表すラベル R_{inter}, C_{inter} が含まれているため，2 つのコード片集合 CF_{G1}, CF_{G2} は互いに言語間チェーンドクローンとなる．

3.3 検出手法

この節では，前節で定義した言語間チェーンドクローンの検出手法について説明する．検出手法はコードクローン検出，依存関係解析，言語間チェーンドクローン検出の 3 つのステップから成る (図 11)．ステップ 1 では入力ソースコードに対してコードクローン検出ツールを適用し，各言語に対応したコードクローンを検出する．そしてステップ 2 では検出されたコードクローン間に存在する依存関係を抽出する．これらの情報を用いて，最後にステップ 3 で言語間チェーンドクローンを検出する．まず始めに手法の対象となる言語やフレームワークについて述べ，その後，各ステップについて詳細を述べる．

3.3.1 対象

本研究で提案する言語間チェーンドクローン検出手法は HTML と JavaScript で記述されたウェブアプリケーションを対象とする．これら 2 つの言語を選択した理由は次の通りである．

1. HTML と JavaScript を組合せたプロジェクト開発が多い．
   対象言語を選択するにあたり，GitHub[1] で公開されているリポジトリに対して，利用頻度の高い言語の組み合わせを調査した．調査ではまず，2014/1/1〜2014/12/31 に作成されたリポジトリから無作為にリポジトリを選択した．そしてその中から複数のプログラミング言語を利用しているリポジトリを抜粋し，相関ルールマイニング [1] を

[1] https://github.com/

表 1: 言語の組み合わせとその出現率 (上位 5 つ)

順位	言語の組み合わせ	出現率 (%)
1	HTML，JavaScript	22.9
2	PHP，JavaScript	6.5
3	C，C++	6.1
4	Ruby，HTML	6.0
5	Ruby，JavaScript	5.6

用いて言語の組合わせの頻度を分析した．その結果，109,547 件のリポジトリの中で 66,825 件のリポジトリが複数の言語を利用しており，その中でも HTML と JavaScript を組み合わせて開発がされているリポジトリが最も多いことが分かった (表 1).

2. ウェブアプリケーションはコードクローンを多く含む.
   Rajapakse らは，ウェブアプリケーションのソースコードを対象にコードクローンの調査を行った．調査の結果，ウェブアプリケーションは，リリース期間の短さが主な原因でコードクローンを多く含むと述べている [15, 16]．言語間チェーンドクローンはコードクローンが存在していることが前提なため，コードクローンを多く含むウェブアプリケーションは検出対象として適していると考えた.

ウェブアプリケーション開発において，開発効率を向上させるためにフレームワークが用いられることがよくある．その中でも MVC フレームワークは，プログラムをアプリケーション中のデータモデル (Model) とそれを画面表示するビュー (View)，そして Model と View を仲介するコントローラ (Controller) の 3 つに分けることでプログラムを簡潔に開発することができる.

JavaScript に対する MVC フレームワークとして，AngularJS[2]・BackboneJS[3]・Ember.js[4] が有名である．その中でも AngularJS はこれら MVC フレームワークの中でも最も広く使われている MVC フレームワークであるため [14]，今回の研究では AngularJS フレームワークを用いたウェブアプリケーションを対象とした検出を行う.

3.3.2 ステップ 1: コードクローン検出

入力ソースコードから HTML と JavaScript の各言語に対応したコードクローンを検出する．コードクローン検出ツールは 2.2 節で紹介した NiCad を利用する．本手法で NiCad を

[2]https://angularjs.org/
[3]http://backbonejs.org/
[4]http://emberjs.com/

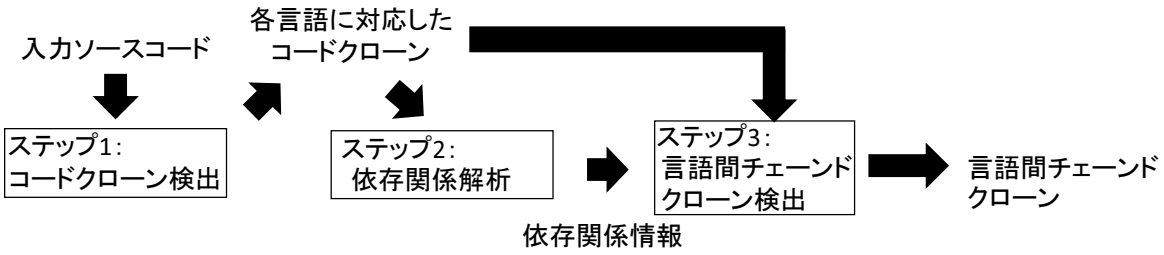

図 11: 言語間チェーンドクローン検出手法の概要図

選択した理由は 2 つある．1 つ目の理由はタイプ 1 からタイプ 3 までのコードクローンを検出するためである．コードクローンに対して実施されるリファクタリングについて調査した研究によると，差異を含むコードクローンに対してもリファクタリングが実施されることが分かっている [6]．このことから，タイプ 1 からタイプ 3 までのコードクローンを検出する必要があると考えた．2 つ目の理由は拡張性の高さである．HTML や JavaScript で記述されたソースコードからコードクローンを検出する専用の検出手法やツールは存在しないため，既存の検出手法やツールを拡張実装する必要がある．NiCad は 2.2 節でも述べた通り，言語に対応した文法定義や変換ルールを記述したスクリプトを用意することでその言語からのコードクローン検出を行うことができる．そのため，低コストで上記 2 つの言語に対するコードクローン検出を実現することができる．

NiCad によるコードクローン検出において，抽出ステップと正規化ステップは言語に依存する．本手法ではこの 2 つのステップを次の通りに行う．

HTML からのコードクローン検出

まず，コードクローン候補となるコード片の抽出について説明する．リファクタリングを実施することを考えると，抽出するコード片の処理内容はまとまっていた方がよい．そこで本手法では，HTML の開始要素から終了要素までをコードクローン候補のコード片として抽出する．また，HTML 要素内の各属性は順序を考慮しない．そのため，属性値が完全に一致しているが各属性の順序が異なる HTML 要素が存在する場合がある．しかし，こうしたケースでは NiCad による比較ステップで不一致行であると判断されてしまう．この問題を解決するために，TXL による pretty-print 機能を用いて 1 つの属性を 1 つの行に記述し直す．この処理を行うことで，属性の順序が異なる要素が存在していても，比較ステップの際に不一致となる行を減らすことができる．また，正規化では属性値に対して abstraction を適用している．この操作により，属性値に使われている識別子が異なるタイプ 2 のコード

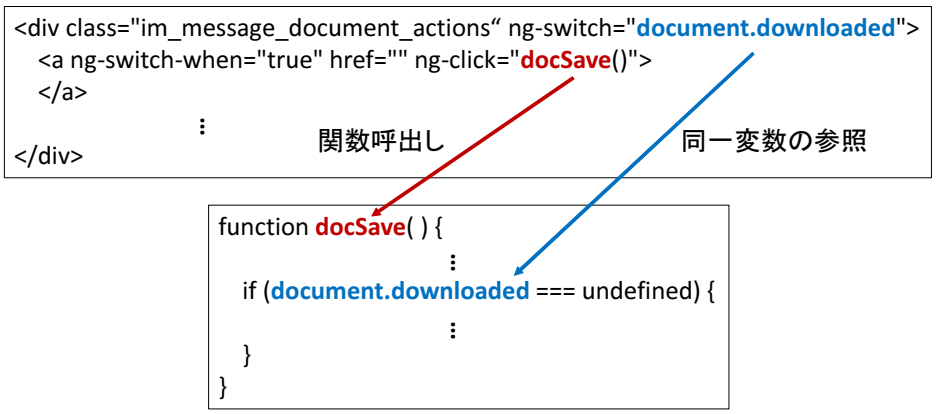

図 12: 抽出される依存関係の例.

クローンを検出することが可能となる.

JavaScript からのコードクローン検出

HTML と同様，抽出されるコード片の処理の内容はまとまっていた方がリファクタリングに都合が良い．したがって，JavaScript の関数 (function) をコードクローン候補となるコード片として抽出をする．また，正規化ステップでは識別子に対して rename を適用している.

3.3.3 ステップ 2: 依存関係解析

ステップ 2 では，検出されたコードクローン同士が持つ依存関係を抽出する．抽出する依存関係は 3.2.1 節で述べた依存関係である．具体的な抽出の方法は，HTML と JavaScript で記述されたソースコードを構文解析し，関数呼び出し文，関数宣言文，変数参照，変数宣言，カスタムディレクティブに利用されている識別子を抽出する．そして 3.2.1 節の依存関係に従って識別子の対応を取ることで依存関係を抽出していく．また，単純に識別子の対応を取るだけではなく，JavaScript の関数に存在しているスコープの解析や AngularJS のコントローラに存在しているスコープの解析も行うことでより正確な依存関係を抽出している.

3.3.4 ステップ 3: 言語間チェーンドクローン検出

ステップ 1 とステップ 2 で得られたコードクローン情報と依存関係情報を用いて，3.2.1 節の定義に従って言語間チェーンドクローンを検出する．これは，コードクローン情報と依存関係情報から構築されるラベル付き有効グラフ全体の中から，

- HTML と JavaScript の両言語で記述されたコード片がノードとなっている．
- 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．ペアを見つける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，依存先/依存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，もしくは，同じクローンセットに含まれていない場合，もしくは，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，HTML と JavaScript の両言語が用いられていれば，その同型部分グラフのペアを言語間チェーンドクローンとして出力する．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．さらに，互いに言語間チェーンドクローンとなっている言語間チェーンドクローン集合を言語間チェーンドクローンセットとして 1 つにまとめ，出力する．

4　ケーススタディ

本稿では，言語横断の依存関係に着目したコードクローン集合である言語間チェーンドクローンを定義した．その目的は，類似機能を実装したコードクローンで構成された集合を開発者に提示することによる，リファクタリング作業の効率化である．

本研究では，実際の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，言語間チェーンドクローンセットが存在するのか，また，類似機能を実装したコードクローンのみで構成された言語間チェーンドクローンセットが存在するのか調査を行う．

4.1　対象

今回提案した言語間チェーンドクローンセットはHTMLとJavaScript，そしてAngularJSフレームワークを利用したウェブアプリケーションを対象としている．そのため，ケーススタディで対象とするプロジェクトもこの条件を満たしている必要がある．今回は，ソースコード共有サービスGitHub上で公開されているプロジェクトに対して，(1) コミット数が1000以上，(2) 関心度を表すスターが100個以上付けられたウェブアプリケーションを無作為に選択し，今回の対象とした．表2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[5]，DuckieTV[6]) 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り，メッセージの送信やグループの作成，チャンネルの開設などができる．また，DuckieTVは，自分の好みのテレビ番組の更新スケジュールをカレンダー上で管理することができるウェブアプリケーションである．

4.2　調査方法

言語間チェーンドクローンセット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言語間チェーンドクローンセットを検出する．コードクローンを検出するステップ1ではコードクローン

表 2: 対象ウェブアプリケーションの規模.

プロジェクト名	総行数 (HTML)	総行数 (JavaScript)
Webogram	6,146	224,140
DuckieTV	3,404	67,485

[5]https://github.com/zhukov/webogram
[6]https://github.com/SchizoDuckie/DuckieTV

表 3: コードクローン検出ツール NiCad に与えるパラメータ.

言語	類似度の閾値	行数の閾値
HTML	0.3	10
JavaScript	0.3	5

検出ツール NiCad を利用している．NiCad は検出時に検出されるコードクローン行数の閾値と類似度の閾値を必要とする．表 3 に今回用いた設定を示した．HTML では開始タグに対応して終了タグが含まれることを考慮して，HTML の行数の閾値を JavaScript の行数の閾値の 2 倍に設定している．

言語間チェーンドクローンセットに含まれるコードクローンが類似機能を実装しているか.

ウェブアプリケーションを実行したときの動作，コードクローン中のユーザ定義名を基に，言語間チェーンドクローンセットに含まれるコードクローンが類似機能を実装しているかを著者の判断で確認する．具体的には，コードクローンへのリファクタリング実施を考えたときに，集約後のコード片に対して集約前のコードクローンの処理内容を反映した名前を付けることができるか，言語間チェーンドクローンセット全体に対してどういった機能を実装しているか名前を付けることができるかという基準で判断をした.

4.3 結果

言語間チェーンドクローンセット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クローンセットの数を表 4 に，言語間チェーンドクローンセットの数とそのサイズを表 5 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

表 4: コードクローン検出ツール NiCad によって検出された HTML と JavaScript に対応するクローンセットの数

プロジェクト名	HTML クローンセット数	JavaScript クローンセット数
Webogram	97	44
DuckieTV	11	65

表 5: 検出された言語間チェーンドクローンセットの数及び，サイズの最大値と最小値

プロジェクト名	言語間チェーンドクローンセット数	最小サイズ	最大サイズ
Webogram	43	4	16
DuckieTV	11	4	12

表 6: 類似機能を実装しているコードクローンのみで構成されている言語間チェーンドクローンセットの数.

プロジェクト名	類似機能を実装しているコードクローンのみで構成されている言語間チェーンドクローンセットの数
Webogram	36
DuckieTV	10

いた．これに対して言語間チェーンドクローン検出を行ったところ，43 個の言語間チェーンドクローンセットが検出された．さらに，言語間チェーンドクローンセット内に含まれるコードクローンの数を調査したところ，最小で 4 個，最大で 16 個のコードクローンが含まれていた．その処理内容について確認したところ，チャットにおける各種転送機能 (動画，ドキュメント，写真，メッセージ) がコードクローンになっていた．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．これに対して言語間チェーンドクローン検出を行うと，11 個の言語間チェーンドクローンセットが検出された．
こちらも同様に，言語間チェーンドクローンセット内に含まれるコードクローンの数を調査した結果，最小で 4 個，最大で 12 個のコードクローンが含まれていた．

今回の研究では言語間チェーンドクローンについて考えているが，言語内の依存関係のみに着目した通常のチェーンドクローンを検出した場合，どのような検出結果が得られるかについても調査した．その結果，HTML チェーンドクローンセットは 2 つのウェブアプリケーションともに検出されなかった．また，JavaScript チェーンドクローンセットについては，Webogram から 9 個，DuckieTV からは 32 個検出された．さらにそれらは言語間チェーンドクローンセットと一切の包含関係を持っていないことも分かった．

言語間チェーンドクローンセットに含まれるコードクローンが類似機能を実装しているか.

検出された各言語間チェーンドクローンセットに含まれるコードクローンが類似機能を実装しているか判断した結果を表 6 に示した．Webogram については，検出された 43 個の言語間チェーンドクローンセットの内，36 個の言語間チェーンドクローンセットで，その中に含まれるコードクローンが類似機能を実装していた．また，DuckieTV の場合は，11 個中 10 個の言語間チェーンドクローンセットが該当していた．図 13 と図 14 に，Webogram から検出された言語間チェーンドクローンセットの中で，類似機能を実装したコードクローンのみで構成されていたケースとそうではなかったケースの例を載せた．図 13 はどちらも，チャットアプリ内に存在するグループとチャンネルに付けることができるタイトルを編集するための UI とその処理を実装している．そのため，タイトルの編集という点で類似機能を実装していると判断した．一方，図 14 の例では，上のコード片がログインしているすべて

のセッションを終了するという機能を，下のコード片が参加人数が大規模なグループへ移行するための処理を実装している．これらは異なる機能であり，コードクローンに対する集約リファクタリングの実行後，適切な名前を付けることが困難であると判断した．また，図13に挙げた言語間チェーンドクローンセットに含まれているコードクローンについて，それらを含むクローンセットのサイズを確認した．その結果，HTMLクローンセットは27個のコードクローンを，JavaScriptクローンセットは4個のコードクローンを含んでいた．さらに，図13のコードクローン以外はすべて異なる機能を実装していることを確認した．

4.4 考察

リファクタリング支援.

表6より，言語間チェーンドクローンセットを構成しているコードクローン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．またその中でも，異なる機能を実装しているコードクローンを多く除外出来ている場合も確認できた．したがって，開発者に言語間チェーンドクローンセットの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によるリファクタリング作業の効率化が期待できる．具体的には，言語間チェーンドクローンセットの場合は2つのコードクローンをチェックするだけでよいが，クローンセットの場合は27個のコードクローンをチェックする必要がある．

実際の運用

表4と表5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言語間チェーンドクローンセットとして検出されなくなっている．これは，依存関係を持たないコードクローンは言語間チェーンドクローン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，検出された言語間チェーンドクローンセット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，開発者にはまず言語間チェーンドクローンセットを見てもらい，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言語間チェーンドクローン検出を組合わせるこうした運用方法が有用であると考えられる．

検出能力の改善

今回のケーススタディでは，言語間チェーンドクローンセット内のコードクローンが類似機能を実装していないケースがみられた．その例を見てみると(図14)，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，検出される言語間チェーンドクローンセット内のコードクローン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

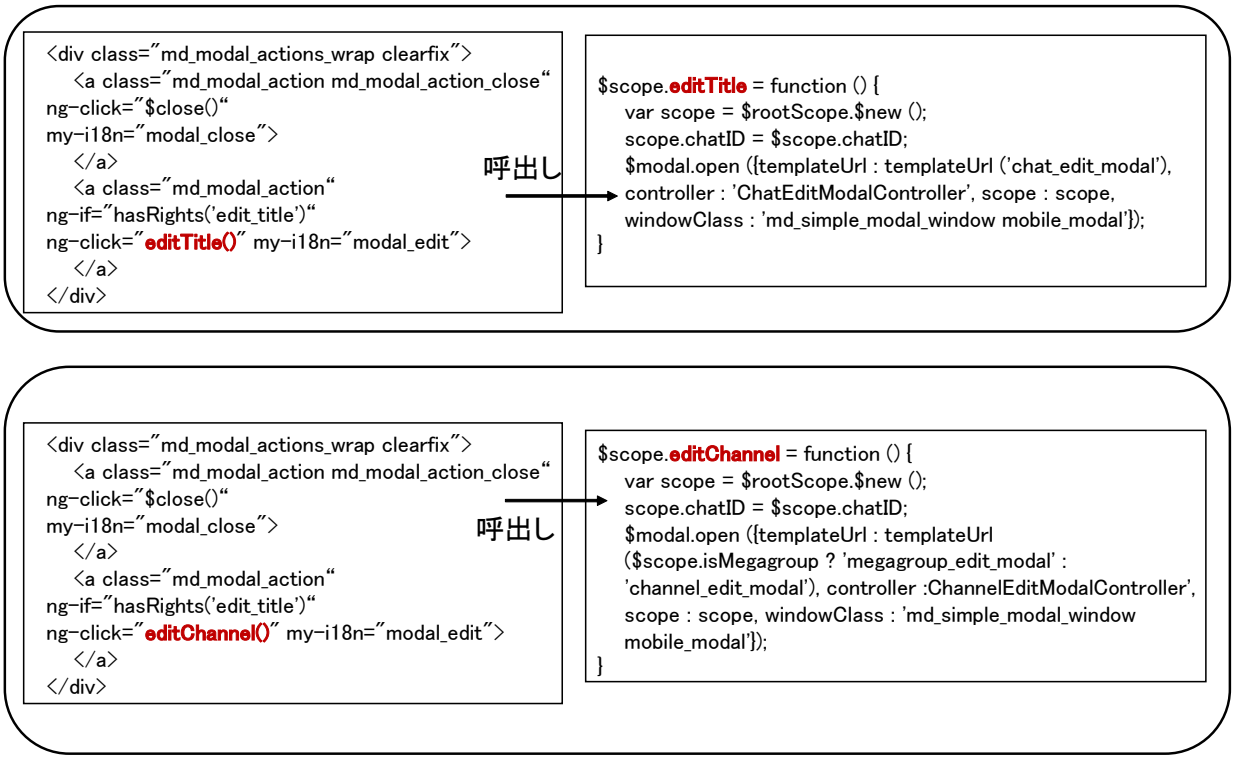

図 13: 類似機能を実装している例．赤字は呼出し関係部分．

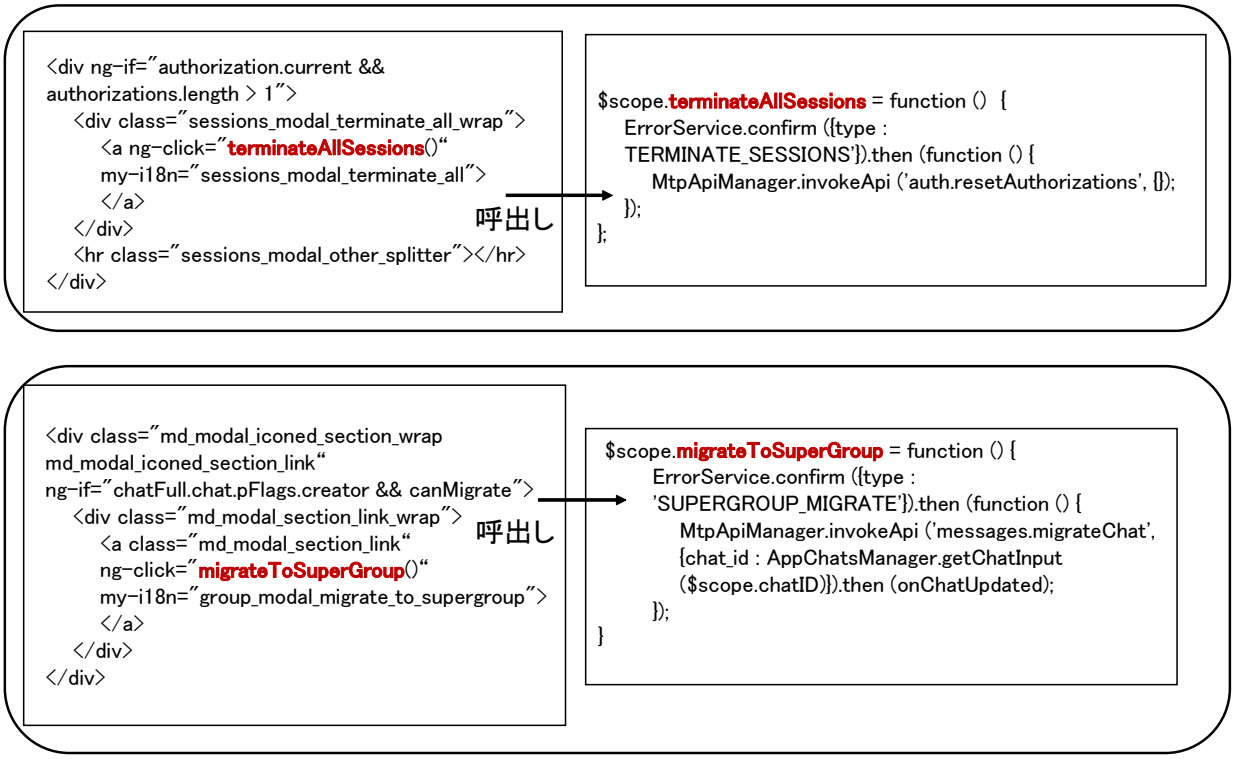

図 14: 異なる機能を実装している例．赤字は呼出し関係部分．

がある．例えば，関数呼出しに使われている識別子が大きく異なる場合には言語間チェーンドクローンとして検出しないという方法が考えられる．

5　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依存関係が存在する．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間でも依存関係を持つことがある．本研究では，こうした言語横断の依存関係を持つコードクローンを言語間チェーンドクローン，その集合を言語間チェーンドクローンセットとして定義し，それらを検出する手法について提案した．言語間チェーンドクローンセットを開発者に提示することで，開発者は類似機能を実装したコードクローンのみをチェックすることができ，類似した機能単位で実施されるリファクタリング作業の効率が向上することが期待できる．

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それぞれ 43 個と 11 個の言語間チェーンドクローンセットを検出した．さらに，言語間チェーンドクローンセット内のコードクローンがすべて類似した機能を実装しているケースが検出された．その中，クローンセットを提示するよりも言語間チェーンドクローンセットを提示する方が類似機能でまとまっていてリファクタリング作業の支援ができるケースも確認された．

今後の課題としては，利用しているフレームワークや言語を変更したときにどのような結果が得られるかという部分について調査を行いたい．また，言語間チェーンドクローンセットを実際に運用していくことも考えている．

謝辞

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 教授には常に適切な御指導及び御助言を賜りました．井上教授の御指導のおかげで本論文を完成させることができました．井上 教授に深く感謝いたします．

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 准教授には研究に関する適切な御指導及び御助言を賜りました．また，普段の生活においても御助言を賜り，有意義な研究生活を送ることができました．松下 准教授に深く感謝いたします．

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 助教には本研究に関する御助言だけでなく，研究室の計算機に関する多くの知識を賜りました．石尾 助教に深く感謝いたします．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田則裕 准教授には本研究において直接御指導賜りました．終始多くの御指導を頂いたおかげで本論文を執筆することができました．吉田 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名修介 特任教授には適切な御指導及び御助言を賜りました．本研究に関する問題点の指摘など，多くの御助言を頂いた春名 特任教授に深く感謝いたします．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 崔恩瀞 助教には，研究の補助など多くの御協力をいただきました．また，学生生活においても私を御気に掛けて下さりました．私が本論文を完成させることができたこと及び充実した生活を送ることができたのは崔 助教のおかげであると，心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった，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研究室の皆様に深く感謝いたします．

参考文献

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pp. 207–216, 1993.

[2] R. Baishakhi, P. Daryl, F. Vladimir, and D. Premkumar. A large scale study of programming languages and code quality in github. In *Proc. of FSE 2014*, pp. 155–165, 2014.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM 1998*, pp. 368–377, 1998.

[4] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9, pp. 577–591, 2007.

[5] K. Roy Chanchal and R. Cordy James. NICAD: Accurate detection of Near-Miss intentional clones using flexible Pretty-Printing and code normalization. In *Proc. of ICPC 2008*, pp. 172–181, 2008.

[6] Eunjgon Choi, Norihiro Yoshida, and Katsuro Inoue. What kind of and how clones are refactored?: A case study of three oss projects. In *Proceedings of the Fifth Workshop on Refactoring Tools*, pp. 1–7, 2012.

[7] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, Vol. 61, No. 3, pp. 190–210, 2006.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[9] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE 2007*, pp. 96–105, 2007.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 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.

[12] N.A. Kraft, B.W. Bonds, and R.K. Smith. Cross-language clone detection. In *Proc. of SEKE 2008*, pp. 54–59, 2008.

[13] Tariq Muhammad, Minhaz F. Zibran, Yosuke Yamamoto, and Chanchal K. Roy. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proc. of CCECE 2013*, pp. 1–6, 2013.

[14] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in javascript mvc applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 325–335, 2015.

[15] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Proc. of ICWE 2005*, pp. 252–262, 2005.

[16] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proc. of ICSE 2007*, pp. 116–126, 2007.

[17] 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.

[18] 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.