マイクロサービスにおける コードクローンの言語間分析

太田 悠希　吉田 則裕　崔 恩瀞　槇原 絵里奈　横井 一輝

マイクロサービスとは，複雑なソフトウェアを相互に通信可能な小規模サービス群に分割するアーキテクチャスタイルである．既存研究おいて，マイクロサービスの各サービスは小規模なプログラムで実現されているにもかかわらず，多くのサービスにコードクローンが含まれていることが報告されている．また，それらコードクローンの同時修正が報告されており，マイクロサービスにおいてコードクローンが保守コストを増大させていることがわかっている．しかし，既存研究が行った調査では，8 個のみのプロジェクトに含まれるサービスを対象としており，それらサービスはすべて Java で開発されている．そのため，様々な言語で開発された多くのサービスを対象とした調査を行うと，Mo らの調査とは大きく異なる結果が得られる可能性がある．そこで本研究では，12 言語で開発された 284 個のプロジェクトを対象としてマイクロサービスに含まれるコードクローンの調査を行った．その結果，C#はプログラム全体におけるコードクローンの割合や，複数のコードクローンが同時に修正されるものの割合が高いことがわかった．

1　はじめに

マイクロサービスとは，複雑なソフトウェアを相互に通信可能な小規模サービス群に分割するアーキテクチャスタイルである [3] [8] [13]．マイクロサービスの特徴の 1 つは，疎結合なサービス群に分割することで，各サービスの開発やデプロイ，保守を独立して行うことができることである [8] [13]．

マイクロサービスを採用したプロジェクトにおいて，モジュール性を考慮しながら小規模サービス群に分割されているのであれば，各サービスに含まれるコードクローンは少ないことが予想される．しかし，Mo らの研究では，対象としたサービスの多くにコードクローンが含まれていると報告されている [8]．また，それらコードクローンの同時修正が報告されており，マイクロサービスにおいてコードクローンが保守コストを増大させていることがわかっている [8]．これら Mo ら研究成果は，マイクロサービスを採用したプロジェクトに対して，コードクローンのライブラリ化を支援するなどの保守支援が必要であることを示唆している．マイクロサービスでは，各サービスを独立して開発できるため，各サービスは Java だけでなく様々な言語で開発されているにもかかわらず，Mo らの調査対象は Java で開発されたサービスのみである．そのため，他言語で開発されたプロジェクトがマイクロサービスを採用していたときに，コードクローンに対して保守支援が必要かどうかはわかっていない．

そこで本研究では，マイクロサービスにおけるコードクローンに関して，対象を 12 言語に拡大し，保守支援の必要性を調査した．具体的には，d'Aragona らが収集したマイクロサービスの大規模データセット [1] を対象として，クローン率や同時修正率に関して言語間分析を行った．本稿において，クローン率はプログラム全体に対するコードクローンの割合を指し，同時修正率は全クローンセット（2.1 節参照）のうちの同時修正されるものの割合を指す．分析対象の 284

Cross-Language Analysis of Code Clones in Microservices

Yuki Ota, Norihiro Yoshida, Erina Makihara, 立命館大学, Ritsumeikan University.

Eunjong Choi, 京都工芸繊維大学, Kyoto Institute of Technology.

Kazuki Yokoi, 株式会社 NTT データグループ, NTT DATA Group Corporation.

個の OSS プロジェクトは，Java を含む 12 言語 †1 で記述されている．

本分析では，以下の 3 つの RQ を設定した．

RQ1： クローン率が高い言語はどれか？

RQ2： プロダクトコードとテストコード間でクローン率に差異はあるか？

RQ3： コードクローンに対する同時修正率が高い言語はどれか？

RQ1 と RQ3 では，各言語で記述されたコードクローンに対して保守支援の必要性を調査するために，各言語のクローン率や同時修正率を計測した．RQ2 では，言語ごとにテストフレームワークが異なることから，テストコードのクローン率や同時修正率が言語間で異なると考え，プロダクトコードとテストコードを分けて計測を行った．

12 言語で記述されたプログラムに対してコードクローン検出を行うため，容易に対応言語を増やすことが可能な検出ツールである CCFinderSW [10] を用いた．CCFinderSW は，CCFinder [6] と同じくトークン列の等価性に基づく Type-2 クローン（2.1 節参照）を検出する．分析結果の概要を以下に示す．

- Java と C#はクローン率と同時修正率の両者が高く，これら言語で記述されたサービスは，コードクローンにより保守性が低下していると考えられる．そのため，コードクローンのライブラリ化などの保守支援が必要であると考えられる．
- プロダクトコードとテストコード間で，クローン率や同時修正率に有意差がある言語が存在した．このため，同一言語であってもテストコードかどうかで，コードクローンが保守性に与える影響は異なると考えられる．このことから，コードクローンのライブラリ化などの保守支援を検討する際は，プロダクトコードとテストコードを区別する必要があると考えられる．

以降，2 章においてコードクローンやマイクロサービスに関する関連研究を述べ，3 章では対象プロジェクトや言語を選定するための予備調査について述べる．その後，4 章と 5 章において，それぞれ分析と結果を説明する．6 章で分析結果と本調査の制限等について考察を行い，最後に 7 章で本稿をまとめる．

2 関連研究

2.1 コードクローンとその検出ツール

コードクローンとは，プログラム中に存在する互いに一致，または類似したコード片を指す [4]．これまでに，トークン列や構文木の照合や深層学習に基づきコードクローンを検出する手法が数多く提案されてきている [5] [6] [9] [10] [12]．互いにコードクローンとなる 2 つのコード片の組をクローンペアと呼び，コードクローンの同値類をクローンセットと呼ぶ．本稿では，以下の 3 つのコードクローンの分類を用いる [9]．

Type-1 クローン 空白やタブの有無，括弧の位置などのコーディングスタイル，コメントの有無などの違いを除き完全に一致するコードクローン

Type-2 クローン Type-1 クローンの違いに加えて，変数名や関数名などのユーザ定義名，変数の型などが異なるコードクローン

Type-3 クローン Type-2 クローンの違いに加えて，文の挿入や削除，変更などが行われたコードクローン

従来，コードクローン検出ツールの対応言語を増加させることが困難であったが，瀬村らは多様なプログラミング言語に容易に対応できるコードクローン検出ツール CCFinderSW を開発した [10]．CCFinderSW は，構文解析器生成系の 1 つである ANTLR で利用される構文定義記述から字句解析に必要な文法を自動的に抽出する．そして，抽出した文法に基づき Type-2 クローンを検出する．CCFinderSW の利用者は，構文定義記述が集められたリポジトリ grammars-v4 †2 から対象言語の構文定義記述を取得し，ツールの実行時に入力として与えることで対応言語を増加させることができる．

†1 C と C++は，1 つの言語として数えている．この理由は，本研究で使用したコードクローン検出ツールである CCFinderSW が，これらを 1 つの言語として扱うからである．

†2 https://github.com/antlr/grammars-v4

2.2 マイクロサービスとそのデータセット

マイクロサービスとは，小さな独立したサービスを組み合わせて 1 つの大きなアプリケーションを構成するアーキテクチャである [3] [8] [13]．その特徴として，サービス間の独立性，疎結合，データ分離を保ちながら開発とデプロイを行うことが挙げられる [8] [13]．これらの特徴が，アプリケーションのスケーラビリティと開発の迅速性をもたらしている．

d'Aragona らは，OSS リポジトリの大規模コレクションである World of Code [7] からマイクロサービスを採用した 387 個のプロジェクトを抽出し，データセットとして公開している [1]．

2.3 Java におけるマイクロサービスのコードクローンの調査

Mo らは，OSS のマイクロサービスプロジェクトに含まれる Type-1 と Type-2，Type-3 クローンの存在とそれらの同時修正を調査した [8]．彼らの調査では，バージョン V_i で検出されたクローンペア C_i が修正され，V_{i+1} でクローンペア C_{i+1} として検出された場合，クローンペア C_{i+1} を同時修正されたクローンペアと定義した．

彼らの調査結果によると，サービス内では 57.1%から 91.7%，サービス間では 35.7%から 87.5%の *LOC* がクローンになっている．さらに，プロジェクトごとに 5 バージョンを比較した結果，サービス内では 28.6%から 60.0%，サービス間では 14.3%から 63.6%の *LOC* が同時修正されたクローンになっていた．

Mo らの調査は，サービス内外のクローンペアを区別した分析を行っている点や，Type-3 クローンを対象としている点において優れているが，その一方で以下に示す 3 つの問題点がある．

- マイクロサービスでは，各サービスを独立して開発できるため，各サービスは Java だけでなく様々な言語で開発されているにもかかわらず，調査対象のサービスがすべて Java で開発されている．
- 調査対象のプロジェクトの数が 8 個のみである．
- テストコードに含まれるコードクローンの一部は，テストフレームワークが原因で生じると考えられるが，プロダクトコードに含まれるコードクローンと区別せず調査されていいる．

表 1　検出対象のプログラミング言語

C/C++, Java, Perl, PHP, Python, Ruby
Rust, Scala, Go, JavaScript, TypeScript, C#

3 予備調査

本章では，分析対象のプロジェクトや言語を選定するために実施した予備調査について説明する．

まず，2.2 節で説明した d'Aragona らのデータセットには，リポジトリが現存しないプロジェクトが 24 個あったため，これらを全て除外した．

次に，プロジェクトにおいて使用されているプログラミング言語を調査した †3．この調査は，以下の手順で実施した．

手順 1： データセットからソースコードを取得する．

手順 2： プロジェクトごとに得られたソースコードに対して GitHub Linguist†4 を実行する．

手順 3： 実行結果から各言語の *LOC*(Lines of Code) を計算．

手順 4： HTML などの非プログラミング言語を除外し，プロジェクトにおける使用言語比率を算出する．

表 2 は，この予備調査で得られた，全プロジェクトにおける言語ごとの LOC を降順に並べたものである．この表に基づき，本研究で対象する言語を表 1 のとおりに定めた．また，関数型言語 Elixir は命令型プログラミングのための言語ではないので除いた．GitHub†5 上で配布されている CCFinderSW は JavaScript，TypeScript，C#に対応していなかったため，構文定義記述を追加することによってこれら言

†3 d'Aragona らのデータセットには，各プロジェクトの使用言語の項目があるものの，一部のプロジェクトにおいて other だけ記載されており使用言語を特定できないプロジェクトがあった．そのため本研究では，この項目は使用しなかった．

†4 Git リポジトリ中の使用言語を特定するライブラリ https://github.com/github-linguist/linguist

†5 https://github.com/YuichiSemura/CCFinderSW

表 2　全プロジェクトの言語ごとの LOC

言語	K*LOC*	言語	K*LOC*
Java	169,647	Rust	33,074
Python	158,332	Ruby	28,558
JavaScript	156,462	Scala	19,390
Go	87,738	C	18,215
PHP	74,280	Elixir	17,671
TypeScript	64,034	Perl	9,918
C++	35,889	Vue	3,875
C#	34,294	Kotlin	1,348

語に対応した CCFinderSW を本研究では用いた．

最後に，プログラム全体のうち対象言語で書かれたプログラムの割合が 95%以上のプロジェクトを抽出した．条件を満たさないプロジェクトが 70 個存在したため，これらを全て除外した．

予備調査の結果として，除外されなかった 284 個のプロジェクトを分析対象に設定した．

4　分析

マイクロサービスにおけるクローン率や同時修正率に関して言語間比較を行うために，以下の RQ を設定した．

RQ1： クローン率が高い言語はどれか？

RQ2： プロダクトコードとテストコードでクローン率に違いがあるか？

RQ3： コードクローンに対する同時修正率が高い言語はどれか？

言語ごとにテストフレームワークが異なることから，テストコードのクローン率や同時修正率が言語間で異なると考え，RQ2 および RQ3 ではプロダクトコードとテストコードを分けて，分析を行った．ファイルのパス名やファイル名に小文字大文字を区別なく `test` が含まれていたら，そのファイルに含まれるプログラムは全てテストコードとして扱う．それ以外のファイルは，全てプロダクトコードとして扱う．また，あるクローンセットに含まれるコード片が 1 つ以上テストコードに含まれていたら，テストコードのクローンセットとする．あるクローンセットに含まれるコード片が全てプロダクトコードに含まれていたら，プロダクトコードのクローンセットとする．

次に，分析手法について説明する．本分析では，3 章で説明した 284 個のプロジェクトを分析対象とする．図 1 は RQ に回答するために実施する本研究の分析手法の概要を示す．この図で示すように，d'Aragona らのデータセットからリポジトリの URL を取得し，`git clone` コマンドを用いて GitHub から分析対象のプロジェクトの最新ソースコードを取得する．

次に，取得したソースコードに対して，CCFinderSW を用いてコードクローンを検出する．このとき，CCFinderSW の設定は，検出範囲をファイル間に，出力形式をクローンセット，検出するコードクローンの最低トークン数 (しきい値）を CCFinderSW のデフォルト値である 50 トークンに設定した．検出範囲をファイル間にしたのは，サービス間や機能間のコードクローンを検出するためである．

その後，本分析のために作成したプログラムが CCFinderSW の検出結果ファイルを読み込み，分析に情報を付加してデータベースに格納する．このとき，作成したプログラムが CCFinderSW の検出結果ファイルから，ファイル情報とクローンセット情報を取得し，それらの情報をデータベースに格納する．また，ファイルごとのクローン率情報もデータベースに格納する．本研究では，クローン率は $ROC(F)$ の値を用いて計算する．$ROC(F)$ はファイル F がどの程度重複化しているかを表す指標である．$ROC(F)$ は以下の式で計算される．

$$ROC(F) = \frac{LOC_{duplicated}(F)}{LOC(F)}$$

上の式で，$LOC(F)$ は F の行数を，$LOC_{duplicated}(F)$ は F の LOC のうちクローンセットに含まれている LOC を示す．これ以降では，$ROC(F)$ をクローン率と呼ぶ．また，クローンセット情報をデータベースに格納するとき，`git blame` コマンドを用いて，クローンセットに含まれているコード片ごとに含まれるコミット情報を付け加える．

最後に，データベースに格納された情報を言語ごと，テストコードかプロダクトコードか同時修正されたクローンセットかどうかに基づいて分類し，各 RQ に回答する．プロダクトコードとテストコードの区別は以下のとおりである．

- ファイルのパス名やファイル名に小文字大文字を区別なく `test` が含まれていたら，そのファイ

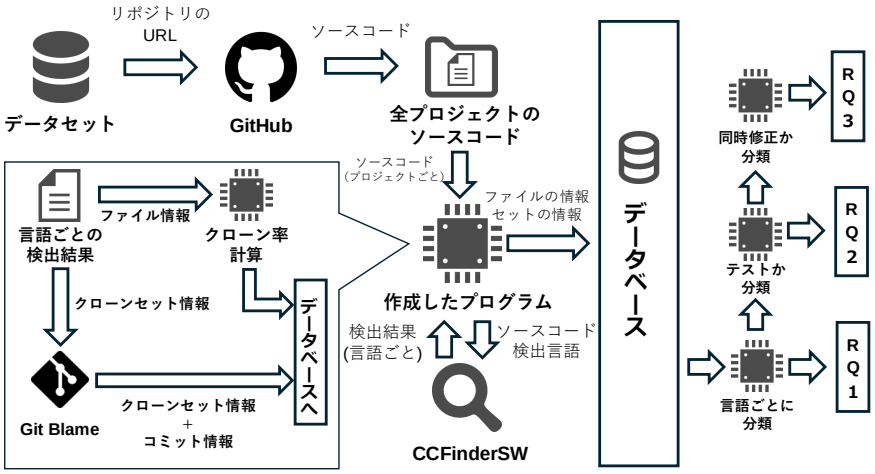

図 1　分析手法の概要

ルに含まれるプログラムは全てテストコードとして扱う．

- それ以外のファイルは，全てプロダクトコードとして扱う．

また，クローンセットがプロダクトコードとテストコードのどちらであるかの判定基準は以下のとおりである．

- あるクローンセットに含まれるコード片が 1 つ以上テストコードに含まれていたら，テストコードのクローンセットとする．
- それ以外（つまり，あるクローンセットに含まれるコード片が全てプロダクトコードに含まれていたら），プロダクトコードのクローンセットとする．

以下の手順で，各クローンセットに対して，同時修正が行われたか判定する．この際，最新のコミットから 100 コミットを分析対象とした．これは，データセット [1] に含まれるプロジェクトは，最低 100 以上のコミットを持つためである．

手順 1： クローンセットに含まれる各コード片に対して，クローン検出したバージョンからみて 100 バージョン以内の修正を列挙し，その中で最も新しい修正を特定する．以上の作業をクローンセットに含まれる全てのコード片に対して行う．

手順 2： 1. で特定した修正集合の中に，同一コミットが含まれていれば，同時修正が行われたと判定する．さもなくば，同時修正が行われなかったと判定する．

5　分析結果

5.1　RQ1: クローン率が高い言語はどれか？

4 章で定義したクローン率に基づき，プロジェクトごとに各言語の平均クローン率を算出し，算出した値を言語間で比較した．その分析結果を図 2 に示す．この図の縦軸はクローン率，横軸は言語を示す．

図 2 が示すように，クローン率の中央値が最も高かった言語は，C#で 40%であった．C#に次いで、Java が 39%、Go が 35%と高い値を示した．また，最小値に着目すると，C#は 12%であるのに対し，Java と Go はそれぞれ 1%と 4%であった．一方，クローン率の中央値が最も低かった言語は，Ruby と C/C++ で，いずれも 10%であった．また，Ruby と C/C++ は最大値はそれぞれ 25%，20%であった．

RQ1 への回答

最もクローン率の中央値が高い言語は C#で 40%であった．2 位と 3 位は Java，Go であり，それぞれ 39%，35%であった．

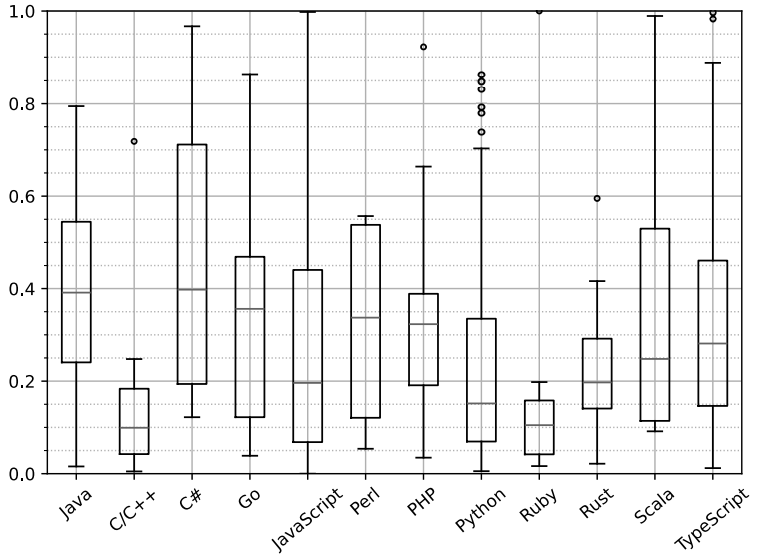

図 2　言語ごとのクローン率の箱ひげ図

表 3　言語ごとのプロダクトコードとテストコードの LOC およびプロジェクト数

（各プロジェクトが複数言語に該当することがある）

言語	KLOC_p	KLOC_t	N_F	N_P
Java	2,745	1,223	30,808	78
C/C++	399	57	2,018	13
C#	565	16	8,890	20
Go	1,419	770	9,888	28
JavaScript	7,515	490	32,457	152
Perl	35	3	309	4
PHP	1,349	177	11,314	23
Python	2,299	1,517	29,908	88
Ruby	690	108	12,848	14
Rust	528	253	4,190	10
Scala	302	215	4,935	10
TypeScript	2,032	293	24,539	54
合計	19,877	5,122	1,721,04	284

5.2　RQ2: プロダクトコードとテストコードでクローン率に違いがあるか？

プロダクトコードとテストコードにおけるクローン率に違いがあるか，言語ごとに分析した．

テストコードとプロダクトコードの言語ごとのクローン率の分析結果を図 3 に示す．この図に示すように，プロダクトコードにおけるクローン率の中央値の上位 3 言語は，C#，Java，Go で，それぞれ 39%，39%，37%であった．また，C#の最小値が 12%である一方，Java と Go の最小値はそれぞれ 0%と 1%であった．テストコードでは，クローン率の中央値の上位 3 言語は，C#，Java，PHP で，それぞれ 37%，37%，29%であった．また，C#，Java，PHP のいずれの言語も最小値は 0%であった．C#と Java はプロダクトコードとテストコードの両方で高いクローン率を示した．

また，各言語ごとのプロジェクト数，および言語ごとのプロダクトコードとテストコードの LOC を表 3 に示す．この表では，LOC_p はプロジェクトコードの LOC の合計，LOC_t はテストコードの LOC の合計，N_F はファイル数，N_P はプロジェクト数を示す．また，プロジェクト数は 1 つのプロジェクトに複数の言語が含まれている場合がある．この表より，いずれの言語でもプロダクトコードとテストコードは十分な量存在することが分かる．

最後に，プロダクトコードとテストコードにおけるクローン率の有意差を，有意水準 5%でマン・ホイットニーの U 検定で確かめた．その結果，Go と Python でプロダクトコードとテストコードの間に有意な差が見られた．

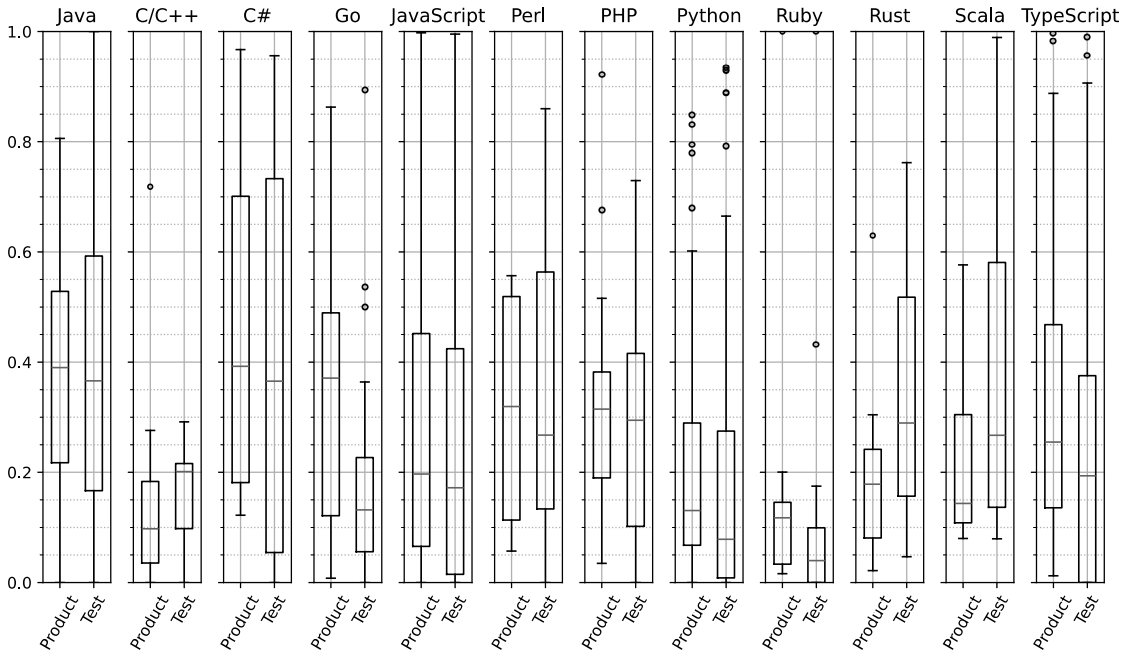

図 3　テストコードとプロダクトコードにおける言語ごとのクローン率

RQ2 への回答

Go と Python で，クローン率におけるプロダクトコードとテストコードの有意差が見られた．したがって，一部の言語では，プロダクトコードとテストコードのクローン率が異なるといえる．

5.3　RQ3: コードクローンに対する同時修正率が高い言語はどれか？

コードクローンに対する同時修正率が言語によって異なるか，テストコードとプロダクトコードを区別し分析した．本分析における同時修正率は，同時修正されたクローンセットの数を全体のクローンセットの数で割ったものである．

分析結果を図 4 に示す．この図では，縦軸にプロジェクトごとの同時修正率を示し，横軸に言語ごとにテストコードとプロダクトコードを分けて示す．図に示すように，C#の同時修正率の中央値が全言語で最も高く，71%であった．一方，C/C++と Perl の同時修正率の中央値が全言語で最も低く，1%であった．これらの結果から，言語によって同時修正率が異なると明らかになった．プロダクトコードにおける同時修正率の中央値の上位 3 件言語は C#，TypeScript，Java で，それぞれ 76%，38%，36%であった．これらの言語の最小値に着目すると，C#は 11%である一方で，TypeScript と Java では 0%であった．また，テストコードにおける同時修正率の中央値の上位 3 件言語は C#，Java，Ruby で，それぞれ 66%，42%，38%であった．これらの言語の最小値は全て 0%であった．

最後に，プロダクトコードとテストコードにおける同時修正率の有意差を，有意水準 5%でマン・ホイットニーの U 検定で確かめた．その結果，TypeScript で有意な差が見られた．

RQ3 への回答

最も同時修正率の中央値が高かった言語は C#で，71%であった．一方，最も同時修正率の中央値が低かった言語は，C/C++と Perl で，1%であった．したがって，言語によって同時修正率が異なるといえる．また，TypeScript において，プロダクトコードとテストコードの間で同時修正率に有意差があった．

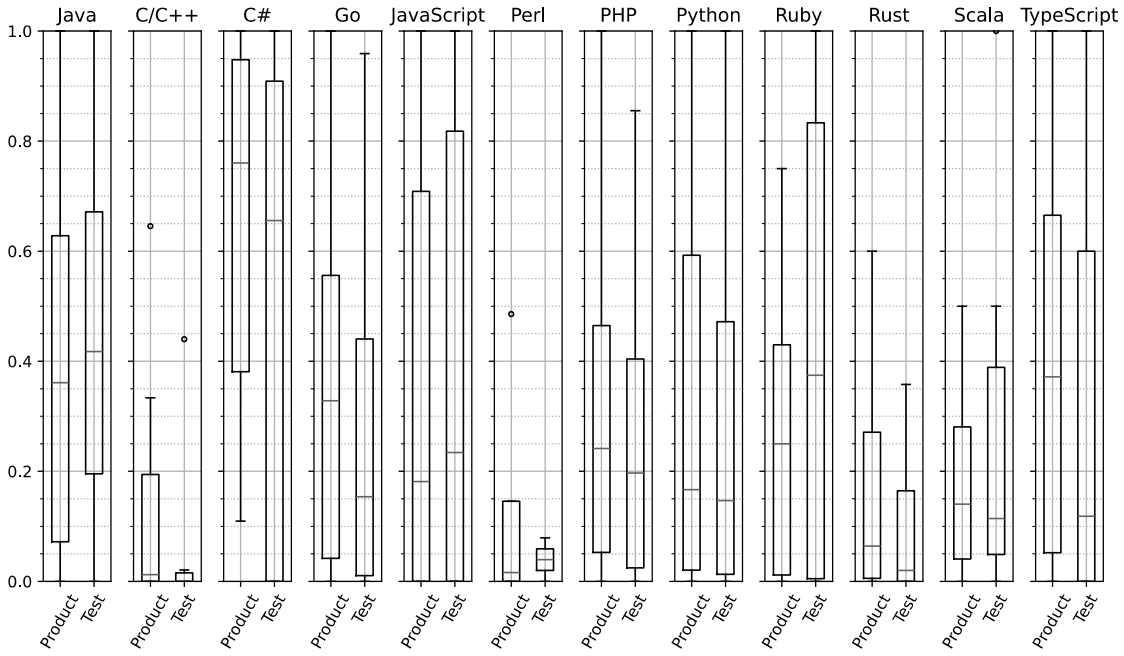

図 4　プロダクトコードとテストコードにおける言語ごとの同時修正率

6　考察

6.1　分析結果に対する考察

RQ2 と RQ2 の結果より，プロダクトコードとテストコード間で，クローン率や同時修正率に有意差がある言語が存在した．RQ2 では，Go と Python において，プロダクトコードとテストコードの間にクローン率に有意な差が見られた．また，RQ3 では，TypeScript において，プロダクトコードとテストコードの間に同時修正率に有意な差が見られた．このため，同一言語であってもテストコードかどうかで，コードクローンが保守性に与える影響は異なると考えられる．よって，コードクローンのライブラリ化などの保守支援を検討する際は，プロダクトコードとテストコードを区別する必要があると考えられる．

分析結果より，どの言語のコードクローンが保守性を低下させるかを考察する．コードクローンの存在はソフトウェアの保守性を低下させる原因の一つとして知られている．しかし，すべてのコードクローンが保守コストを増大させるわけではない．コードクローンに変更が発生した際に，変更が伝播することで保守性を低下させる．既存研究では，プロジェクトの生存期間の間に一度も変更されない，あるいは定常的に変更されないコードクローンが存在することを示している [11] [2]．したがって，クローン率と同時修正率が高い言語が保守性を低下させる．5.1 節より，プロダクトコードでは，クローン率の中央値の上位 3 言語は，C#，Java，Go で，それぞれ 39%，39%，37%であった．また，5.3 節より，プロダクトコードにおける，同時修正率の中央値の上位 3 言語は C#，TypeScript，Java で，それぞれ 76%，38%，36%であった．これらの結果から，C#と Java のクローン率と同時修正率が共に高く，これらの言語のコードクローンが保守性を低下させる．そのため，コードクローンのライブラリ化などの保守支援が必要であると考えられる．

6.2　妥当性への脅威

はじめに，コードクローン検出ツールについて考える．本分析では多様なプログラミング言語に一様に対応するため，クローン検出ツールに CCFinderSW を用いた．しかし，このツールでは Type-2 までのコー

ドクローンしか検出できない．したがって，Type-3のコードクローンの分析は本分析では行っていない．

次にクローン検出範囲について考察する．Mo らの調査では，コードクローンをマイクロサービス内とサービス間に分類して調査した．しかし，Mo らの調査と比べ，本分析ではデータセットのプロジェクト数が増加したため，クローンセットをサービス間とサービス内のものに手動で分類することが困難であった[8]．したがって，本分析の結果は，サービス内クローンおよびサービス間クローンについて調査した結果ではない．

次に自動生成コードについて考察する．本分析ではテストコードとプロダクトコードを分けて分析した．よって，テストコードの関係しているクローンセットによる結果への影響は考慮できている．しかし，RPCシステムの一種である gRPC[†6] や，オブジェクト指向言語の getter/setter などのプロダクトコードに含まれうる自動生成コードが存在する．通常，自動生成コードの分類は，ソースコードのコメントの分類によって行われる．今回の分析では，多数の言語の分析を行うため，その言語ごとに異なった手法で分類を行う必要がある．したがって，本分析の結果には自動生成コードによる結果への影響は考えられていない．

次にテストコードの定義について考察する．本分析では，クローンセットに 1 つでもテストコードが含まれていれば，テストコードのクローンセットとして扱った．しかし，プロダクトコードとテストコードを分けてクローン検出を行う手法も存在する．したがって，その場合の結果は本分析では考えられていない．

次に言語ごとの特徴について考察する．本分析ではクローンの検出範囲をファイル間に限定している．しかし，Java であれば 1 ファイルに 1 クラス，Python であれば 1 ファイルに 1 モジュールなど，言語ごとに 1 ファイルに含まれうる単位が異なる．したがって，言語ごとのファイル構造の違いに起因する結果の変動は考慮できていない．また，プロジェクト内での用いられ方が言語ごとに異なる場合がある．さらに，言語ごとに用いられるフレームワークが異なっている．しかし，本分析ではコードクローンの発生要因の分析は行なっていない．したがって，言語ごとの用いられ方やフレームワークによる結果への影響は考慮できていない．

最後に，同時修正の定義について考察する．本分析では同時修正の判定をコミット番号で行っている．しかし，コミット自体を複数に分けた上で単一マージで取り込んで整合性を取る場合は同時修正と判定できていない．また，本分析では先行研究[8] とは異なり，最新版のクローンのみを同時修正の判定に用いている．これは，大規模なデータセットに対応するためである．したがって，本分析の結果はこれらによる影響は考えられていない．

7 まとめ

本稿では，Mo らの調査結果を踏まえて，多言語かつ大規模な OSS のマイクロサービスのデータセットに対するコードクローンの分析を行った[8]．予備調査ではデータセットのプロジェクトにどのような言語が含まれるかを調査した．分析はプロジェクト数が多いデータセットに対して処理を行い，RQ ごとに言語，テスト，同時修正の有無で分類することで行った．その分析結果は，マイクロサービスでは言語ごと，テストコードとプロダクトコードでコードクローンが異なった性質を持つことを示した．また，Java と C#のコードクローンが他の言語と比べ保守性を低下させることも示した．

今後の展望として，以下が挙げられる．

- Mo らの調査では Type-3 クローンを検出し調査を行っていた．しかし，本分析では，対象とする言語を増やしたことによって，それに対応できる Type-3 クローン検出ツールが存在しなかった [†7]．したがって，そのようなコードクローン検出ツールが開発された際の課題とする．
- マイクロサービスでは，サービス間のコードク

†6 https://grpc.io/

†7 多様な言語に対応可能な Type-3 クローン検出ツールとして MSCCD[14] が挙げられるが，本分析で対象とした Perl に非対応であり，Scala や Ruby のプログラムに対してコンパイルエラーが発生することある．

ローンが独立性を脅かす．Mo らの調査ではクローンペアをサービス内とサービス間に分類していた．しかし，本分析ではデータセットの規模が大きくなったことで分類が困難になり，行えなかった． そこで，大規模なデータセットにも対応可能な分類手法を今後考案する．

- 自動生成コードがプロダクトコードのコードクローンの結果に影響を与えていることが考えられる．しかし，本分析では自動生成コードを分類した分析が行えていない．したがって，多言語なデータセットにも対応可能な自動生成コードの分類手法を今後考案する必要がある．
- 本分析は，プロダクトコードとテストコードを分けてクローン検出を行っていない．したがって，プロダクトコードとテストコードを分けてクローン検出を行った結果の分析を今後行う．
- 本分析の考察では C#と Java のコードクローンが他の言語よりも保守性を低下させると結論付けた．しかし，この言語による結果の違いが何に起因するかは分かっていない．したがって，言語による結果の違いの原因を今後調査する．

謝辞 本研究は，JST さきがけ JPMJPR21PA ならびに JSPS 科研費 JP24K02923，JP20K11745，JP23K11046 の支援を受けた．

参 考 文 献

[1] d'Aragona, D. A., Bakhtin, A., Li, X., Su, R., Adams, L., Aponte, E., Boyle, F., Boyle, P., Koerner, R., Lee, J., Tian, F., Wang, Y., Nyyssölä, J., Quevedo, E., Rahaman, S. M., Abdelfattah, A. S., Mäntylä, M., Cerny, T., and Taibi, D.: A Dataset of Microservices-based Open-Source Projects, *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, 2024, pp. 504–509.

[2] Göde, N. and Koschke, R.: Frequency and risks of changes to clones, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA, Association for Computing Machinery, 2011, pp. 311–320.

[3] 本田澄, 大八木勇太朗, 井垣宏, 福安直樹: マイクロサービス開発入門者のための簡易な教材の開発, 第 *9* 回実践的 *IT* 教育シンポジウム論文集, (2023), pp. 83–92.

[4] 井上克郎, 神谷年洋, 楠本真二: コードクローン検出法, コンピュータソフトウェア, Vol. 18, No. 5(2001), pp. 47–54.

[5] Jiang, L., Misherghi, G., Su, Z., and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[6] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 07(2002), pp. 654–670.

[7] Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., and Mockus, A.: World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 143–154.

[8] Mo, R., Zhao, Y., Feng, Q., and Li, Z.: The existence and co-modifications of code clones within or across microservices, *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–11.

[9] Roy, C. K., Cordy, J. R., and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, Vol. 74, No. 7(2009), pp. 470–495.

[10] 瀬村雄一, 吉田則裕, 崔恩瀞, 井上克郎: 多様なプログラミング言語に対応可能なコードクローン検出ツール CCFinderSW, 電子情報通信学会論文誌 *D*, Vol. 103, No. 4(2020), pp. 215–227.

[11] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹: コードクローン変更管理システムの開発と実プロジェクトへの適用, 情報処理学会論文誌, Vol. 54, No. 2(2013), pp. 883–893.

[12] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X.: A Novel Neural Source Code Representation Based on Abstract Syntax Tree, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.

[13] Zhao, Y., Mo, R., Zhang, Y., Zhang, S., and Xiong, P.: Exploring and Understanding Cross-service Code Clones in Microservice Projects, *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 449–459.

[14] Zhu, W., Yoshida, N., Kamiya, T., Choi, E., and Takada, H.: MSCCD: grammar pluggable clone detection based on ANTLR parser generation, *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, New York, NY, USA, Association for Computing Machinery, 2022, pp. 460–470.